Joining and aggregating datasets using CouchDB

Zach Smith

9 April 2018

Abstract

Data mining typically requires implementing operations that involve cross-cutting entity boundaries and are awkward to implement in document-oriented databases. CouchDB, for example, models entities as documents, with highly isolated entity boundaries, and on which joins cannot be directly performed.

This project shows how join and aggregation can be achieved across entity boundaries in such systems, as encountered for example in the pre-processing and exploration stages of educational data mining. A software stack is presented as a means by which this can be achieved; first, datasets are processed via ETL operations, then MapReduce is used to create indices of ordered and aggregated data. Finally, a Couchdb list function is used to iterate through these indices and perform joins, and to compute aggregated values on joined datasets such as variance and correlations.

In terms of the case study, it is shown that the proposed approach to implementing cross-document joins and aggregation is effective and scalable. In addition, it was discovered that for the 2014 - 2016 UCT cohorts, NBT scores correlate better with final grades for the CSC1015F course than do Grade 12 results for English, Science and Mathematics.

Acknowledgements

I would like to give special thanks to my project supervisor Associate Professor Sonia Berman - an extremely patient supervisor who had the foresight to push me (at great effort) in the direction that resulted in a completed MSc. I would also like to thank her for the effort she put in in terms of data acquisition and guiding me in designing the research topic. I would like to thank Jane Hendry and Stephen Marquard for providing the data exports.

I would like to thank my family: Craig Smith, Meryl Smith, Josh Smith and Stephanie Honchell for showing support and understanding of the time I have dedicated to this over project. On a similar note I would like to thank my colleagues at Avesta (particularly Patrick DeSomma and Matthew Fluharty) for patience and understanding over the course of this project.

And lastly I would like to especially thank Stephanie Honchell and Patrick DeSomma for helping me with insights, editing and reading. And also Guy Bedford, Jan Lehnardt, Robert Newsom, Rudelee Merks, Nick Kuilman, Greg Van Berkel, and many others who helped me in very many ways.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the implementation and usage of big data expands, alternatives to traditional relational-orientated databases are becoming the preferred software for housing large data stores. SQL systems such as Oracle, DB2, SQL Server, MySQL and others, tend to scale better vertically (using more powerful computers) rather than horizontally (using more computers networked together) [20]. This can be limiting and expensive when compared to NoSQL databases, which allow for the seamless expansion from single to multiple servers. But swapping out relational-storage for newer alternatives involves a mental shift at many levels within the software stack; this is most evident at the data-retrieval layer, with the shift from using SQL (structured query language) to query data stores to other paradigms that are less familiar to most data professionals (software engineers, software architects, database administrators, etc.). Although many NoSQL databases (databases that don't implement a relational model) implement a version of the SQL standard for querying (which eases the learning curve for new technologies), many do not.

One alternative paradigm to SQL is MapReduce, a logical framework for data querying that allows for the efficient processing of dispersed datasets. This technology, which is already used by several software giants [16], is being adopted as part of the new technology stack as part of the ongoing trend of "information explosion". As distributed computing power becomes more obtainable through the proliferation of cloud providers such as Digital Ocean, Hetzner, Amazon, Google, etc., it is worth investigating how technologies that make use of dispersed processing via the MapReduce paradigm can do relational operations such as joins and aggregations.

Also relevant to the shift from relational to non-relational databases is the increasing diversity of the data being collected digitally. As mentioned by the Couchbase

project [20] and observed from general experience in the modern workplace, much of the data produced on a day-to-day basis is semi-structured or unstructured (for example, text documents and spreadsheets). In turn, increasing technological gains, such as those represented by the proliferation of IOT (internet of things) devices, require the housing of ever more varied digital data. RDBMSs seem ungainly in this scenario, with their strictly defined data models making it cumbersome to handle semi-structured and unstructured data. Appropriate systems are expensive in terms of implementation times and complex in terms of architecting and usage. Storing data without having to first define rigid models allows for a more agile approach to data modeling. Additionally, as a system evolves, subsequent changes to unstructured data models become straightforward and the knock-on effects of code-changes are much more isolated.

1.1 Project Significance

CouchDB [5] is a new database that embodies much of the NoSQL trend: it has a schema-less data model; it accomplishes data processing via MapReduce; it exists as an open-source code-base, and it is suitable for distribution over commodity hardware.

With a focus on highly available data and a replication API implemented across multiple types of devices (servers, browsers, tablets, mobile phones, etc.), CouchDB provides a suitable foundation for building offline-first applications that can be used in relatively disconnected locations [3]. For example, CouchDB was deployed as part of the effort to contain the 2013 - 2016 Ebola outbreak, providing a means of digital data collection in areas with unreliable internet [3]. Similarly, there is a lot of scope for offline-first application development in the South African context, where data is still very expensive and internet access remains sporadic throughout much of the country.

Data mining, the extraction of knowledge from data, is a typical big data use case in a growing number of fields, including education. An understanding of the relation-

ship between indicators such as student grades, online participation, engagement with assigned materials, or demographic information can allow for more adaptive pedagogical approaches to teaching and learning. As such, data mining the massive amount of information that learning management systems (LMS) – such as the University of Cape Town's (UCT) Sakai platform [50] – collects has the potential to greatly improve the educational experience. Systems that support data-storage and retrieval are integral to the process of working with these datasets and the logical frameworks that guide the design of such systems are becoming increasingly important as the rate and amount of data collected continues to grow exponentially.

This thesis investigates a method of joining and aggregating data using CouchDB in the context of Educational Data Mining (EDM). These methods are validated through a case study exploring the effectiveness of UCT's student profiling during the admissions process in relation to performance in the first-year Computer Science course, CSC1015F. In addition, the correlation of students' usage of the Sakai LMS in relation to their relative position in class is also calculated.

This project is conducted under ethical approval granted by UCT. All information that can be used to personally identify individual students (such as student numbers) is anonymized, and at no point has any data been publicly accessible.

1.2 Motivation & Aim

Theoretically, CouchDB is suitable for storing an unlimited amount of unstructured data across distributed clusters of commodity servers [2]. Its API, that is effectively an interface for the manipulation of b+tree structures is a novel approach to data handling and working with the database directly from an HTTP client is incredibly convenient. Such features make CouchDB a suitable tool for the information-orientated society of the future, where an agile approach to data storage will, by necessity, become the norm as the increasingly interconnected world will produce more and more unstructured data needing to be processed.

In short, CouchDB is an innovative database that allows for innovative systems. Case studies involving CouchDB are therefore necessary to develop an understanding of all the different use-cases that such novel software can represent.

The aim of this project is to devise methods to perform cross-document joins and aggregations using CouchDB, and to apply this in the context of an educational data case study.

Chapter 2

Background

Any data-driven investigation requires learning of a variety of tools and concepts. This chapter covers:

- A conceptual overview of MapReduce as a paradigm for working with data, and its specific implementation in CouchDB
- A high-level description of the CouchDB storage engine and approaches to structuring/modeling data in a document-orientated database
- A brief description of CouchDB's HTTP API

2.1 MapReduce

In response to dealing with huge amounts of data daily, authors at Google (Jeffrey Dean and Sanjay Ghemawat) outlined the MapReduce framework as a means of abstracting the complications associated with distributed computing such as data distribution, fault tolerance, load balancing, and how to parallelize processing [21]. MapReduce provides programmers with a conceptually-simple interface for specifying dispersed data computations succinctly and hides the implementation details.

The framework relies on a simple programming model described by [21] as a computation that takes a set of input key-value pairs and produces a set of output key-value pairs using the following 3 steps:

- A *mapping* stage in which distributed key-value pairs are produced from input data as described by a user-defined map function
- A *grouping* stage where distributed key-value output from the mapping stage is collected to common keys - i.e. *key:[value, ..., value]* datasets

- And a *reducing* stage where values per grouped key are processed as described by a user-defined reduce function

Due to the distributed and isolated nature of the map and reduce tasks, MapReduce is fault tolerant (specifically, fault tolerance is implemented via reexecution), which has in turn resulted in the *New Software Stack* [46]: large-scale computing clusters built on commodity (cheap) hardware and software that computes in parallel. Such an approach to building and maintaining systems represents the potential to process ever-greater amounts of data at ever cheaper rates. This has spurred information explosion across all manner of software applications.

Starting with the development of the Hadoop framework as an open-source alternative to Google's proprietary file system and MapReduce framework, MapReduce implementations have become mainstream. Companies that make use of this programming model include Yahoo, Facebook, Amazon, and many more [16]. The Apache Foundation maintains a list of companies that use the Hadoop framework [7].

Use-cases for systems utilizing MapReduce overlap significantly with business domains that have long been cornered by RDBMSs. As such, a great deal of research has focused on leveraging MapReduce as a means of implementing operations that are traditionally associated with RDBMSs - specifically in instances of *relational-algebra* [16, 46]. The full spectrum of relational operations: *selection*, *projection*, *union*, *intersection*, *difference*, *joins* (exempting non-equi joins, which cannot be implemented via MapReduce), *grouping* and *aggregation* are translatable to MapReduce [46]. MapReduce allows for implementing *joins*, including both *Two-way* and *Multi-way*, by using a variety of algorithms. A master's thesis from the University of Edinburgh demonstrates joining in Hadoop via *Map-Side joins*, *Reduce-Side One-Shot joins*,*Reduce-Side Cascade joins* and other algorithms [16].

2.2 CouchDB

In CouchDB, data is modeled conceptually as *documents*, which users interact with as serialized JSON strings [14]. There are no limitations, specifications, etc., on how documents may be structured (other than that they must be represented by valid JSON).

The CouchDB data model (like many NoSQL data models) is considered *unstructured* or *semi-structured* when compared to RDBMSs, where relations within data models are rigidly formalized. But despite using different storage models, the conceptual requirements of data-persistence – such as ACID properties – need to be addressed. ACID properties provide certain guarantees that are required within a system in order to assure information persistence:

- *atomicity*: a series of operations that are part of the same logical transaction should either succeed or fail completely
- *consistency*: building on the concept of atomicity, the result of a single logical transaction should leave the database in a working and valid state
- *isolation*: Transactions, whether run concurrently or sequentially, are deterministic. That is, transactions on a database are performed in isolation of each other
- *durability*: data is permanently persisted, irrespective of subsequent system failures

Without such guarantees a database would not be useful since the integrity of any information stored by such a system would be questionable. In terms of the CouchDB software, ACID properties are applicable at the document level, meaning that the interactions with individual documents are either successful or not; CouchDB does not allow the reading or writing of partial documents.

CouchDB serializes document interactions to guarantee isolation and document writes are guaranteed to be durable and result in a consistent database. In other words, in working with CouchDB, transactions involving single documents are completely reliable and fault tolerant; however, transactions involving retrieving and

storing information from multiple documents are not (although bulk document insertions can also be configured to be atomic).

Single-document ACID guarantees make multistep transactions in CouchDB more difficult. *Multi-step transactional atomicity* is a key feature for many RDBMSs including MySQL, SQL Server, etc., and overcoming this limitation is required in order to implement NoSQL databases within traditional RDBMS environments. This is possible, both in CouchDB specifically [47] and in NoSQL generally, [34] via the implementation of ACID/transactional properties as external bespoke middleware that engages with the database management systems (DBMS). But this is not an ideal alternative to systems where RDBMS-specific features are required. This is a trade-off that CouchDB (and other NoSQL) databases make in favor of less rigorous data models that are suitable for use-cases that RDBMSs are not; namely, environments with semi-unstructured data where availability is more important than complete consistency.

CouchDB documents are written to disc using an *append-only* algorithm, in which documents are incorporated into a b+tree structure, which is the database [35]. Such structures facilitate rapid data retrieval from huge databases. Additionally, CouchDB incorporates *Multiversion concurrency control* (MVCC) into the storage engine as a means of versioning nodes within the tree. Aside from the serialization of write operations, MVCC negates the need for locks, which are typically implemented within RDBMSs and are expensive in terms of computer resources. Without locking on read/write operations, CouchDB data stores are always available, with the caveat that retrieved documents, as found through searching the tree structure, may not always be the most recent versions of those documents. This can be the case if a new version of a document is written whilst the same document is being retrieved elsewhere.

For systems faced with the potential of network downtime, research in 1999 introduced the idea of the CAP theorem as a trade-off analysis that can be applied to 3 properties within a data storage system (Consistency, Availability, and Partition-tolerance) [26]. CouchDB represents a trade-off of *consistency* for *availability* in

order to facilitate scalability in terms of handling large amounts of data in nearly real time. CouchDB emphasizes availability and the eventual consistency of data to allow for a high partition tolerance of databases with unreliable communication between nodes. From CouchDB 2.0 onwards, users can configure the level of consistency for document reads by specifying a quantifiable amount of confidence that retrieved data is not out of date. It should be noted, however, that this is not possible for document writes.

Because of the potential for inconsistency, CouchDB seeks to provide a 'relaxed' viewing model - i.e. a *soft state*, where data representation is not tied to the underlying entities. As part of the trade-off of availability at the expense of consistency, data conflicts - where entities are updated separately and independently of each other - are often acceptable in NoSQL databases, particularly in systems that are required to take an *offline-first* approach to data-handling. Thus, these systems provide a means of interacting with partial datasets, which, when synchronized, could result in a conflicting state.

MVCC allows for the handling of such consistency violations by maintaining consistency within the data storage layer itself, which is separate from the consistency of information. Inconsistent (i.e. conflicting) information can be resolved by following the audit trail created by the MVCC mechanism. By maintaining a database state that is separate from the information-state, multi-master replication in highly-available systems that lack strong consistency guarantees is possible.

CouchDB is usually selected as a database over alternative JSON stores (such as MongoDB) for its replication capabilities - that is, for the ease with which multi-master, fault-tolerant clusters can be set up. The separation of the database state from the information greatly facilitates this. Compared to replication within relational systems (such as in SQL Server), here replication is independent of information (i.e. table structure, relationships between tables, etc.) and, as a result, is much simpler to work with.

2.2.1 Entity Modeling

Despite NoSQL DBMSs moving away from the relational model provided by RDBMSs, data models still focus on the modeling of entities. Depending on what these entities constitute, NoSQL databases can be grouped into two categories [25]:

- *aggregate-orientated* stores that model data similarly to the relational model, but with isolated entity boundaries and
- *aggregate-ignorant* stores, wherein the concept of entities is fundamentally different (e.g. a graph database such as Neo4J [49], where the entities are edges and nodes)

Most databases operate within a domain where data is, for the most part, entity-driven. There are hundreds of NoSQL data stores and a comprehensive catalog of such products is not necessary [17], but familiar examples within the family of aggregate-orientated NoSQL databases include key-value stores such as Amazon's Dynamo [49]; column based stores such as Apache's Cassandra [49] or HBase [49] (as part of the Hadoop ecosystem); and document stores such as CouchDB or Mongo [49].

Although NoSQL databases are said to be *schema-less*, this is something of a misnomer [9]: NoSQL databases allow for inconsistent schema representation across different entity instances. Such flexibility is at the heart of document stores such as CouchDB and Mongo, where loose-schema modeling is one of the properties that makes such technologies suitable for large systems that generate data from inconsistent sources.

JSON structure allows for forming hierarchical (tree) structures of infinite depth - i.e. for nesting child entities as sub-objects. As an alternative to relationships that are defined by key references, hierarchical structures allow for the easy structuring of specific entities but are less suitable for working with classes of entities. As such, data is typically grouped differently in JSON documents as compared to relational tables; objects encourage groupings of specific entities, while tabular relations encourage the grouping of entity types.

Using hierarchical object structures, it is easy to implement compositional object models, where entities are encapsulated exclusively within parent entities. Aggregational and associative relationships (i.e. *many-many* associations), however, require that data be denormalized, which results in replicated information. This is inefficient but advantageous for dealing with entities whose instances have varying data structures. For example, it is conceivable that the instance of a *Person* entity could have different fields, with similar fields holding different types of data (e.g., a person may have a single address or a list of addresses).

To think of entities as rows within a relational table, each entity represented via JSON is allowed a unique list of columns featuring differing data types as values. In a relational database this would result in a sparse table where every row must include the columns of every other row of the same entity type. Such a case would result in a massive amount of wasted storage and inefficient indexing.

Entity modeling in CouchDB is quite flexible. RDBMSs allow for the categorization of information in terms of relations, fields, constraints, and other objects, while MongoDB documents are identifiable in terms of *collections*. CouchDB does not provide a built-in means of classifying documents, except by scanning an entire database and reading each document.

Entity classification can be enforced by including a field *type* that allows for entity modeling to be conducted separately from the data-storage engine. With this approach, documents are only classifiable on retrieval and deserialization. Alternatively, entity classification can be included in the *_id* field of CouchDB documents, which has the benefit of allowing for indexed entity retrieval.

2.2.2 API

HTTP Interface

All communication with CouchDB takes place via the HTTP protocol. The interface is resource orientated and strives to be RESTful. This makes such interactions very

clear since the HTTP endpoints are logical and easy to remember. For example, the following sample endpoints (among many) are shown here, with the full API documentation available online [4].

- *GET* /dbName/:id (retrieves a document with the specified ID)
- *PUT* /dbName/[:id] (inserts a document, optionally specifying an ID)
- *POST* /_bulk_docs (inserts multiple documents - atomicity of batch insert is configurable but defaults to false)
- *POST/GET* /_all_docs (Fetches multiple documents, specifying keys can be done in in the body of a post request)

Design Documents

CouchDB allows users to specify several different types of functions that can be executed on the server-side Erlang application via JSON - as a regular database document, but with an *_id* of "_design/<name>". Such documents are known as *design* documents and are treated as special by the CouchDB application. CouchDB executes functions that are defined in design documents on the server - functions can be defined in a variety of languages, including JavaScript. There are 6 types of functions that users can define for server-side execution:

- *views*
- *shows*
- *lists*
- *updates*
- *filters*
- *validations*

Views are b+tree indices created by specifying a mapping function and an optional reduce function to be executed on every document in a database. Map functions iteratively take each document as an argument and output key-value pair(s) - as specified by the map function definition - that are then incorporated into the b+tree. Reduce functions are evaluated by taking Map-function outputs as inputs and re-

turning reduced results from that input, which are stored as internal nodes on the b+tree.

Show functions act as a type of middleware, allowing users to specify transformations on a single document that has been requested from the database and then returning that document to the user - for example, transforming a document from JSON representation to HTML representation for better display in a browser. List functions are similar to show functions, but they transform sets of documents that have been iteratively retrieved from an index instead of single documents that have been retrieved from a database.

Update functions allow document updates to be performed indirectly via the HTTP API (functionally equivalent to retrieving a document, updating it, and then reinserting it). Filters, true to their name, allow users to specify filters that can be used during view retrieval, document retrieval, and in replicating data between CouchDB databases. Validations allow users to specify rules regarding when a database's documents can be updated and by whom.

2.2.3 Indices (MapReduce)

Essentially, CouchDB provides users a simple-to-use HTTP interface that allows for fine-grained control over b+tree structures. Documents can either be retrieved from trees directly or indices can be built as representations of databases using MapReduce and then retrieved. Index creation involves iteratively processing every document in a database and outputting a new b+tree structure (and index), providing a *view* of the underlying database. Views provide a means for applying selections, projections, and aggregations of databases.

CouchDB's implementation of MapReduce is quite different from implementations by Google, Hadoop and most other systems. CouchDB incorporates map function output directly into a b+tree, and so does not have a grouping (shuffling) stage. Since one of the functions of the shuffling stage is to ensure that reducers get all map output associated with a particular key at the same time, this is not guaran-

teed in CouchDB. Instead, reducers act on portions of map-output (partial indices) partitioned on boundaries that can fall between index values with the same key. As such, reducers are configured to store intermediate results within the index produced by the map function and employ the concept of *rereduction* in which intermediate values are re-reduced. As a result of this implementation the methods of doing join and aggregation via MapReduce proposed in [46] are not applicable to CouchDB. This MapReduce implementation is comparable Couchbase's approach, which is well explained online [19].

Map functions must be specified by a user and are always executed externally to the main Erlang process via marshalling between the main Erlang process and the indexing engine. Reduce functions, however, can either by executed via the main Erlang process (by specifying a built-in reduce function) or externally by the indexing engine when specifying custom reduce functions. CouchDB automatically includes a MapReduce engine (*couchjs.exe*), which is a JavaScript query-server (indexing engine) coupled with Mozilla's SpiderMonkey runtime engine. This query engine is replaceable by dropping in alternative implementations in JavaScript or other languages, if required. CouchDB spawns a single couchjs.exe process per shard and executes map/reduce functions in the context of new documents sequentially according to the order in which the database was changed [30, 42].

Conceptually, fetching all documents of type x requires specifying an iteration through every document in the database and fetching documents with content indicative of type x. View indices are built into CouchDB in this fashion, via iterating over every document in the database, and passing that document to the user-defined map function. A user writes code in this map function that evaluates each document and emits key-value pairs. The map output is then grouped by key and passed to the reduce function for reduction.

Users can specify whether they want to retrieve reduced results or access the unreduced map results (meaning, a reduce function isn't actually required to produce a CouchDB view). However, when passing output from the map function to the reducer, where map output is grouped by key, there is no guarantee that all the

map output for a particular key will be sent to the same reduce function [6]. As such, a reduce function may repeatedly operate on the same output as the map function, necessitating the 'rereduction' of already reduced output. The reduce function requires handling in cases where `rereduce=true` and `rereduce=false` within the same function body. Therefore, writing custom reduce functions that adhere to the reduce function contract is fairly difficult for anything other than the simplest of examples. CouchDB's reduce function contract is as follows:

- *keys*: a list of tuples of the form of $<key, id>$. *key* is the key emitted by the map function defined by the user and *id* is the ID of the document that was processed by the map function in order to emit the key (this is implicit and not defined by a user). This argument is null in the case of `rereduce=true`.
- *values*: a list of the values emitted by the map function as defined by the user, with each value correlating to the respective element in the keys list (when `rereduce=false`). When `rereduce=true`, this argument is a list of values that were output previously by this same reduce function on an earlier execution.
- *rereduce*: a boolean field indicating whether the function is invoked with output from the map function (`rereduce=false`) or with previous output from the reduce function (`rereduce=true`). Reduce function results are cached on internal nodes in the b+tree view indexes to facilitate incremental tree updates without requiring the recalculation of all reduce output [31], making the rereduce contract necessary.

In the current CouchDB release (as of January 2018), there is an additional means of querying CouchDB databases called 'Mango', which is a selection-based syntax inspired by MongoDB. The Mango syntax still processes documents via MapReduce, but it is generally faster than JavaScript functions since the MapReduce engine can be executed directly within the Erlang process on the server.

_sum Reduce Function

Adherence to the contract of the _sum reduce function requires that values output by the map function are either numerical or a list of numerical values - examples of

both allowable map function output formats are shown in Figure 2.1 A. The reduce function receives values grouped by key (Figure 2.1 B) as the 2nd argument during execution. The reduce function signature is shown in Figure 2.1 C; the 1st argument is a separate list of grouped key-id pairs for every document that outputs a specific key, and the indices of the 1st argument correspond to indices in the 2nd argument (list), in terms of which document produced which value.

Figure 2.1 D shows the logical treatment of the grouped values by the _sum function; for a list of values of the same key, values at corresponding indices are summed. In the case where an index is out of bounds (when grouped values differ by list length, or when some values are numerical and others are lists), the value '0' is used. Output of the _sum function is shown in Figure 2.1 E.

```
/* A: Map output*/
{["key"]: 7} // (_id: x)
{["key"]: [3,1,3]} // (_id: y)
{["key2"]: [2,2,2]} // (_id: z)

/* B: Map output grouped by key */
{["key"]: [7,[3,1,3]]}
{["ke2"]: [[2,2,2]]}

/* C: Map output passed to _sum function as input */
reduce([key:id tuples], [values groupted by key], rereduce)
reduce([["key", "x"], ["key", "y"]], [7,[3,1,3]], false)
reduce([["key2", "z"]], [[2,2,2]], false)

/* D: Logical treatment of values argument (arg 2) during reduction */
{["key"]: [sum([7,3]), sum([0,1]),  sum([0,3])]}
{["key2"]: [sum([2]), sum([2]),  sum([2])]}

/* E: Reduce output (group = true) */
{
  ["key"]: [10, 1, 3],
  ["key2"]: [2,2,2]
}
```

Figure 2.1: MapReduce logic using *_sum*-function

_stats Reduce Function

To adhere to the _stats function contract, key-value output by the map function should follow specific constraints:

- All values must be a single number
- Or, all values must be an array of numbers (each array must be the same length). An example of this format is shown in Figure 2.2 A

This is unlike the contract for the _sum function, where value output is *normalized* during the reduction of lists of unequal length or the reduction of lists and numerical values. The aggregation of grouped values (as shown in Figure 2.2 B) is completed by the _stats function. For values that are groups of numbers (i.e. where the map function outputs key-number) then a single _stats object is output for each unique key. Map output is received as input to the _stats reduce function within 3 parameters as shown in Figure 2.2 C:

- A list of *MapOutputKey*:*Document ID* key-value pairs (all the keys are the same, but the documents that *emitted* them are different)
- A list of values for a particular key
- true / false (rereduction)

The values (argument 2 for the reduce contract) are aggregated by grouping them at common indices and reducing to a single *_stats object* as shown in Figure 2.2 D. (Or, if the map output was a single number and not a list of numbers, the output becomes an aggregation of those numbers). If the values as output by the map function are lists of lists (of numbers), then a list of _stats objects is output by the _stats function as shown in Figure 2.2 E (one _stats object per index within the list). The object output by the _stats function for each aggregated number includes a count of how many items are included in the aggregation, the minimum value, the maximum value, the sum of all the values, and the sum of the squares of all the values. The output of the _stats function includes output of both the _sum and _count built-in reduce functions. However, the stricter contract (when all values are lists of the same length or are numerical) makes this function cumbersome to use

when only a sum or count is required.

```
/* A: Map output*/
{["key"]: [1,1,0]} // (_id: x)
{["key"]: [3,1,3]} // (_id: y)
{["key2"]: [2,2,2]} // (_id: z)

/* B: Map output grouped by key */
{["key"]: [[1,1,0],[3,1,3]]}
{["ke2"]: [[2,2,2]]}

/* C: Map output passed to _stats function as input */
reduce([key:id tuples], [values groupted by key], rereduce)
reduce([["key", "x"], ["key", "y"]], [[1,1,0],[3,1,3]], false)
reduce([["key2", "z"]], [[2,2,2]], false)

/* D: Logical treatment of values argument (arg 2) during reduction */
{["key"]: [aggregate([1,3]), aggregate([1,1]),  aggregate([0,3])]}
{["key2"]: [aggregate([2]), aggregate([2]),  aggregate([2])]}

/* E: Reduce output (group = true) */
{
  ["key"]: [
    {"sum":4,"count":2,"min":1,"max":3,"sumsqr":10},
    {"sum":2,"count":2,"min":1,"max":1,"sumsqr":2},
    {"sum":3,"count":2,"min":0,"max":3,"sumsqr":9}
  ],
  ["key2"]: [
    {"sum":2,"count":1,"min":2,"max":2,"sumsqr":4}
  ]
}
```

Figure 2.2: MapReduce logic using *_stats*-function

Chapter 3

Source Data

The focus of this project is the manipulation of data using CouchDB and associated technologies, whilst specifically using data from the business domain of Educational Data Mining (EDM). This field is well developed and many studies have sought to model student performance based on markers such as attendance, assignment and test grades, high school marks, demographic data, etc. Different means of model generation have been discussed by the EDM community, such as predictive analysis via decision tree generation [11, 15, 22, 29, 37, 45], with varying results. Other models have been applied within the field of EDM, as discussed in a review of EDM up to 2009 [10].

UCT's Chief Information Officer (CIO) Jane Hendry and Stephen Marquard (the Learning Technologies Coordinator from the Center for Innovation in Learning and Teaching at UCT), made anonymized student data available for use in the present study. This encompasses UCT admissions data, course grades, and Sakai interactions. The three separate datasets were recieved in CSV format and are classified as:

- *grades*: UCT course results
- *admissions*: Student Grade 12 and National Benchmark Test (NBT) results
- *events*: Student interactions with the Sakai platform, measured as browser interactions

3.1 Grades

Three years of data are available (2014, 2015, and 2016) and were received as separate CSV files, then compiled into a single file with normalized delimiters. The fields in the compiled file are described in Table 3.1, and a sample of this file is shown in Figure 3.1. Essentially this data comprises a list of the grades achieved for specific courses in specific years, along with the anonymized student numbers. The individual files could have remained separate, but for the purposes of this investigation it is easier to work with a single file rather than multiple files.

Table 3.1

Grades data fields

Field Name	Data type	Description
DownloadedDate	date	Excel date format
RegAcadYear	number	Year
RegTerm	number	Integer ID
anonIDnew	number	Student number
RegProgram	string	Program abbreviation
RegCareer	string	Academic level
Degree	string	Degree code
DegreeDescr	string	Degree title
Subject	string	Three letter abbreviation
Catalog.	string	Catalog sub-component[1]
Course	string	Full course code description
CourseSuffix	string	Course session identifier
Session	string	Session name
Percent	string	Grade achieved by student
Symbol	string	Symbol achieved by student
UnitsTaken	number	Total units taken by student
CourseID	number	Numerical course Identifier
CourseDescr	string	Description of course
CourseCareer	string	Academic level of course
Faculty	string	Course faculty
Dept	string	Faculty department
MaximumCrseUnits	number	N/A
CourseCount	number	N/A
CourseLevel	number	N/A
CESM	number	N/A
Sub-CESM	number	N/A

[1] Course number and session identifier

DnldDt	RegAcadYear	RegTerm	ID	RegProgram	RegCareer	Degree	DegreeDescr	Subject	Catalog.	Course	Suffix	Session
txt	2016	1161	1	CB003	UGRD	QCB007	txt	MAM	1000W	MAM1000W	W	Full Year
txt	2016	1161	1	CB003	UGRD	QCB007	txt	CSC	1015F	CSC1015F	F	Semester One
txt	2016	1161	2	CB004	UGRD	QCB002	txt	CSC	1015F	CSC1015F	F	Semester One
txt	2016	1161	3	EB022	UGRD	QEB028	txt	EEE	2036S	EEE2036S	S	Semester Two
txt	2015	1151	3	EB022	UGRD	EB28	txt	CSC	1015F	CSC1015F	F	Semester One
txt	2016	1161	4	CB004	UGRD	QCB002	txt	CSC	1015F	CSC1015F	F	Semester One
txt	2015	1151	4	CB003	UGRD	CB07	txt	CSC	1015F	CSC1015F	F	Semester One
txt	2015	1151	5	CB003	UGRD	CB07	txt	CSC	1015F	CSC1015F	F	Semester One

Percent	Symbol	UnitsTaken	CourseID	CourseDesc	CourseCrr	Faculty	Dept	MaxCrseUnits	CrseCount	CrseLevel	CESM	Sub-CESM
71	2+	36	107088	txt	UGRD	SCI	MAM	36	1	41	1501	1
70	2+	18	103034	txt	UGRD	SCI	CSC	18	0.5	41	606	6
55	3	18	103034	txt	UGRD	SCI	CSC	18	0.5	41	606	6
63	2-	12	1642	txt	UGRD	EBE	EEE	12	1	42	809	9
77	1	18	103034	txt	UGRD	SCI	CSC	18	0.5	41	606	6
54	3	18	103034	txt	UGRD	SCI	CSC	18	0.5	41	606	6
39	F	18	103034	txt	UGRD	SCI	CSC	18	0.5	41	606	6
39	F	18	103034	txt	UGRD	SCI	CSC	18	0.5	41	606	6

Figure 3.1: Grades data sample

3.2 Admissions

Upon receipt, the files containing data from the three years (2014, 2015, 2016) were formatted inconsistently: corresponding fields were spelled/capitalized differently across the files, the files had completely different sets of fields, the ordering of the fields was inconsistent, and the files were delimited differently. It is possible to work with these files directly but doing so would complicate the analytical procedure and would not add any value to the investigation. As such, a single CSV was compiled addressing the following:

- Field names were adjusted to feature uniform spelling with normalized whitespace (where field names contain whitespace)
- Duplicated fields were removed (UCT performance is appended to each student's admissions data for subsequent years of enrollment)
- Potentially contentious data (race, gender, etc.) that this project is not ethically cleared to work with were removed

The single, processed admissions CSV file contains an anonymized list of students that enrolled at UCT along with corresponding Grade 12 results and NBT scores. The CSV file is described in Table 3.2 and a sample of the file is shown in Figure 3.2.

3.3 Events

Sakai usage data are available for 2016 only and were received as a single CSV file that is 5.2GB in size. Due to its large size, the file cannot be opened in Microsoft Excel and so no scrubbing/amendment was performed on it. Fields in this file are described in Table 3.3. The file as received doesn't contain header information, which had to be obtained separately; for ease of reference, these headers are included in the CSV sample shown in Figure 3.3 [1].

[1] Lines from files can be printed to BASH terminal windows via the command: `head -n <no. of lines> <filePath>`

Table 3.2

Admissions data fields

Field Name	Data type	Description
anonIDnew	number	Anonymized student number
Career	string	Academic level
Citizenship Residency	string	Student citizenship
SA School	string	School name (if in RSA)
Eng Grd12 Rslt	string	Grade 12 English
Math Grd12 Rslt	string	Grade 12 Math
Mth Lit Grd12 Rslt	string	Grade 12 Math Literacy
Adv Mth Grd12 Rslt	string	Grade 12 Advanced Math
Phy Sci Grd12 Rslt	string	Grade 12 Science
NBT AL Score	string	NBT
NBT QL Score	string	NBT
NBT Math Score	string	NBT
RegAcadYear	number	First registration at UCT

Table 3.3

Events data fields

Field Name	Data type	Description
event_date	datetime	When event occurred
event_id	number	Type of event (eg. login)
uct_id	number	Anonymized student number
site_key	number	Foreign key reference to course site[1]
ref	string	Detail of event

[1] This key references Sakai-specific fields and not grades data recieved reparately

ID	Career	Citizenship Residency	SA School	Eng Grd12	Math Grd12	Mth Lit Grd12	Adv Mth Grd12	Phy Sci Grd12	NBT AL Score	NBT QL Score	NBT Math Score	RegAcadYear
1	UGRD	SA Citizen	St Anne's College	84	88			94	80	76	89	2016
2	UGRD	SA Citizen	St Andrews Girls' School	76	78			76	75	58	61	2016
3	Second Year	C	Clifton College	75	78			84	0	0	0	2015
4	Second Year	C	Crawford North Coast College	82	85			94	73	71	86	2015
5	Second Year	C	Ignore	82	85			94	73	71	86	2016

Figure 3.2: Admissions data sample

event_date	event_id	uct_id	site_key	ref
1/6/2016 11:13:18 AM	281	3045582	5401	/presence/192bdcbd-f604-4494-9a9b-3fa602333b3d-presence
1/6/2016 11:13:20 AM	281	2939634	30512	/presence/8f2abfb4-fd78-43bb-bd63-40b5cc3871cb-presence
1/6/2016 11:13:25 AM	281	2933318	16602	/presence/4e168959-8ac0-40c0-a0f2-dc16a315bd48-presence
1/6/2016 11:13:43 AM	281	3004534	4459	/presence/14a625b8-99d6-46d9-8690-cbfc31b7a402-presence
1/6/2016 11:13:45 AM	281	3006684	18894	/presence/5922686b-8b49-4529-8a82-2bb3e11dae1b-presence
1/6/2016 11:13:56 AM	281	2762064	17926	/presence/544d2135-e1cc-4f25-8869-ca35688e9c04-presence
1/6/2016 11:13:59 AM	281	2820866	45041	/presence/d3d13f65-1cd7-4d50-a53d-26c11617e37e-presence
1/6/2016 11:14:01 AM	281	2884124	2723	/presence/0c5f41df-0169-4095-b8f3-dc0e7bb2960b-presence
1/6/2016 11:14:06 AM	281	2884124	21919	/presence/67a4d56d-d6ef-48c7-a207-420094cfab92-presence
1/6/2016 11:14:12 AM	281	2848240	6098	/presence/1c8a2336-ab3a-445c-aad9-a1b96f63e9ef-presence
1/6/2016 11:14:12 AM	281	2884124	6933	/presence/202b7cd1-70a9-4fe9-ae81-57eb9d41232d-presence
1/6/2016 11:14:15 AM	281	2929368	21653	/presence/6655872d-f96b-49a4-869a-7f32f9be827a-presence
1/6/2016 11:14:29 AM	281	2653788	32925	/presence/9aadfa0a-cab3-44e4-9266-2cfcd83115fa-presence

Figure 3.3: Events data sample

Chapter 4

Extraction, Transformation, Load

Extraction, Transformation, and Loading (ETL) processes in the context of this project refers to loading information from CSV files into memory, transforming representation of the information from CSV format to JSON format, and then inserting that information into CouchDB. Once persisted the data is available for exploration by retrieval from the data store directly, or from customized representation of the underlying data store via CouchDB MapReduce views (indices).

Within the context of RDBMSs, there are a large variety of tools available to assist in ETL processing of CSV source data into databases - for example *Open Studio for Data Integration* [53], Microsoft's *SSIS* [36] and many, many more options including a host of cloud-based tools such as *zapier* [54] that allow for data-exchange between a plethora of different platforms. Zapier, for example, allows inserting data into databases from a Google Sheets spreadsheet. But these tools are not available for CouchDB and so bespoke scripting is performed instead. Initially, scripts developed for this project comprised ETL logic for specific source files and transformations specific to file contents. But this approach quickly became unmaintainable due to the difficulty in making changes to scripts where a lot of code was repeated. As such, a configurable ETL tool was developed and incorporated into the project.

The resultant ETL tool is called *Node.js ETL* (nETL) and is designed in terms of *Tasks* that are configured as a pipeline of components (modules). These tasks comprise sequential piping of output from one module to input of another module. Components, by adhering to specified input/output signatures can be strung together in any order allowing for versatile and configurable ETL pipelines.

nETL's core design philosophy is a decoupling of the logic required to manage tasks

from the logic of the tasks themselves. Because the logic of running ETL tasks is separate from the more specific logic of how extractions, transformations and loading logic is applied to a data source, modules that perform extractions, transformations and loading are defined separately to the main code base and can be loaded into nETL during runtime. In other words nETL is authored as a framework in which custom ETL logic is executed on a task-by-task basis.

4.1 Design

Conceptualizing a single ETL process as an entity of type *Task*, that is, the "extraction, transformation and loading of data from a source to destination", provides a focal point on which the nETL software can be architected. Task instances are created by a constructor that takes a configuration object (loaded from a JSON file) as an object. Each task-instance is referenced by a singleton object created on app instantiation - the `taskManager`.

Starting the long-running nETL process comprises instantiating the `taskManager` singleton. This object provides a CLI (command line interface) to facilitate user interactions. Via the CLI, users can interact with `taskManager` to register different ETL components, start/stop tasks, configure application options such as log output path, etc.

The relationship between the `taskManager` singleton, `task` instances, and components is shown in Figure 4.1. ETL components comprise modules that adhere to the Module interface and that accept a TaskConfig object as a paramater on instantiation. As such, modules are paired with configuration objects as shown in the coloured blocks. An ETL task comprises the following steps:

- Loading all component modules required for a specific task into the nETL runtime
- Loading a configuration object that directs how each module should perform into the nETL runtime

- The runtime engine reads the configuration object, and runs the modules as directed

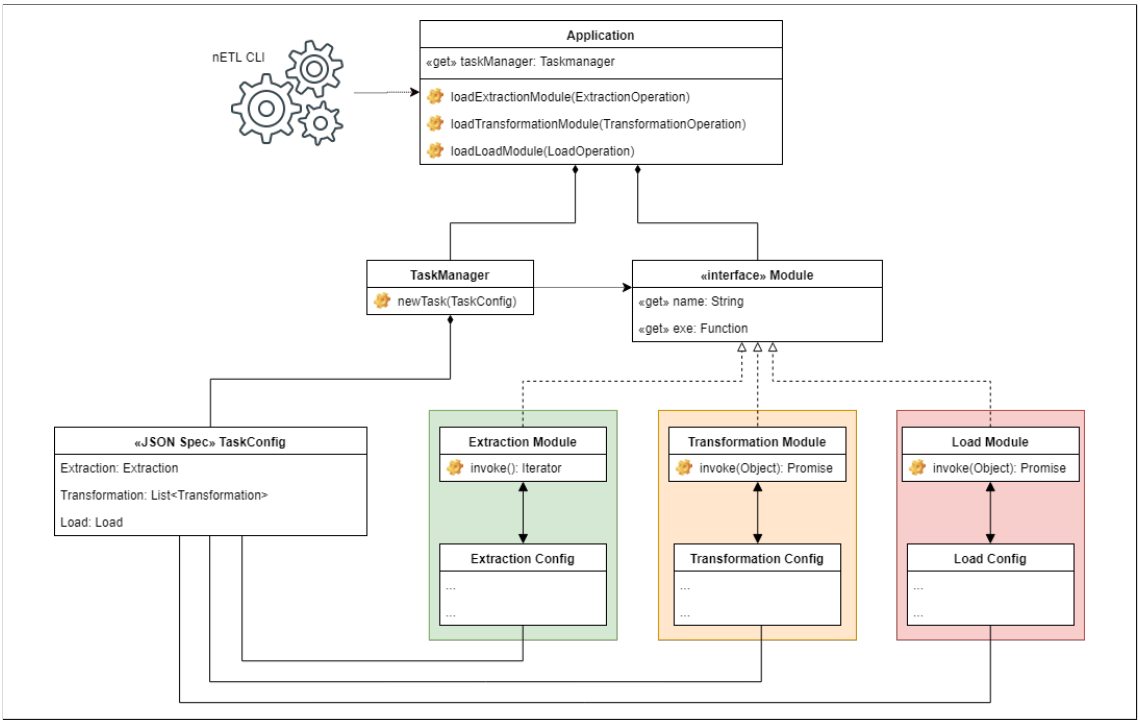


Figure 4.1: nETL architecture

4.2 Implementation

The ETL engine and modules (extractions, transformations, loads) are implemented in JavaScript (*ECMAScript 2017*®) [23] and designed to run in the context of the *node.js* runtime environment [43]. *node.js* emphasizes asynchronous input/output (IO), making it a good fit for handling ETL tasks in which IO (CouchDB is accessed exclusively via network requests) accounts for the greatest amount of computational overhead. Since *node.js* runs server-side it provides access via JavaScript to the file system, which is required in terms of an ETL tool. JavaScript is a sensible language in which to implement nETL in the context of this project:

- It has a very succinct API making it fast to write code in (i.e. it is a highly abstracted language similarly to Ruby or Python)
- But unlike Ruby or Python (and other high level languages), it is opinionated

in that it handles IO asynchronously by default

- The JavaScript implementation of object-orientation is appealing (to some developers at least [52])

Modules are implemented via the *revealing module* pattern, in which functions are returned from functions; parent functions create scoped execution environments that is similar in concept to instantiating classes[1]. Compared to working with constructors closure provides more isolated scope. For each level of closure an additional scope is created that is available solely to functions defined within that scope. Figure 4.2 shows code in which three scoped environments have been defined: a global scope, an Immediately Invoked Function Expression (IIFE) scope, and an *exe*-function level scope (all of these are closed over by an *invoke* function). JavaScript has *lexical* scope [51] that (for this purposes of this project) allows for configuring modules differently at different levels in object hierarchy without having to re instantiate modules that have already been loaded.

[1] JavaScript has classes, but these are just syntactic sugar implemented partially via closure

```
// Global scope

// Closure over the global scope
(function() {
    // Module-specific scope

    // Closure over module-specific scope
    function exe(configurationObj) {
        // Task-specific scope

        // Closure over task-specific scope
        function invoke() {...};

        return {
            invoke: invoke
        };
    };

    return {
        name: "MODULE_NAME",
        exe: exe
    };
})();
```

Figure 4.2: Scoping via JavaScript closure

Modules are loaded into the nETL engine by invoking a function (an IIFE in terms of Figure 4.2) that returns an object with a *name* property for identification and that references the *exe* function. Loading many modules into the application results in a list of available module names, with each name in turn referencing a function named *exe* (these are different functions with the same name). This is shown in Figure 4.3 as the list with the heading *Module Definitions*.

When a task specifies that a module should be used (identified by a name), A lookup for that name in the Module Definitions list is performed and the corresponding *exe* function is executed. This returns a scoped execution context[24] in which another function called *invoke* is required to be defined (nETL users author the body of the *exe* function). The task assigns the name of the module to the *invoke* function definition, which during task-execution may be called many times to perform extraction, transformation, or loading logic. A list of *Loaded Modules* is maintained for

each running task as shown in Figure 4.3. Each of the referenced functions maintains closure over the execution context created on *exe* invocation, thus allowing for a module to be loaded once but configured for specific tasks; every time an *exe* function is called a new closed scope is created only accessible to the callee.

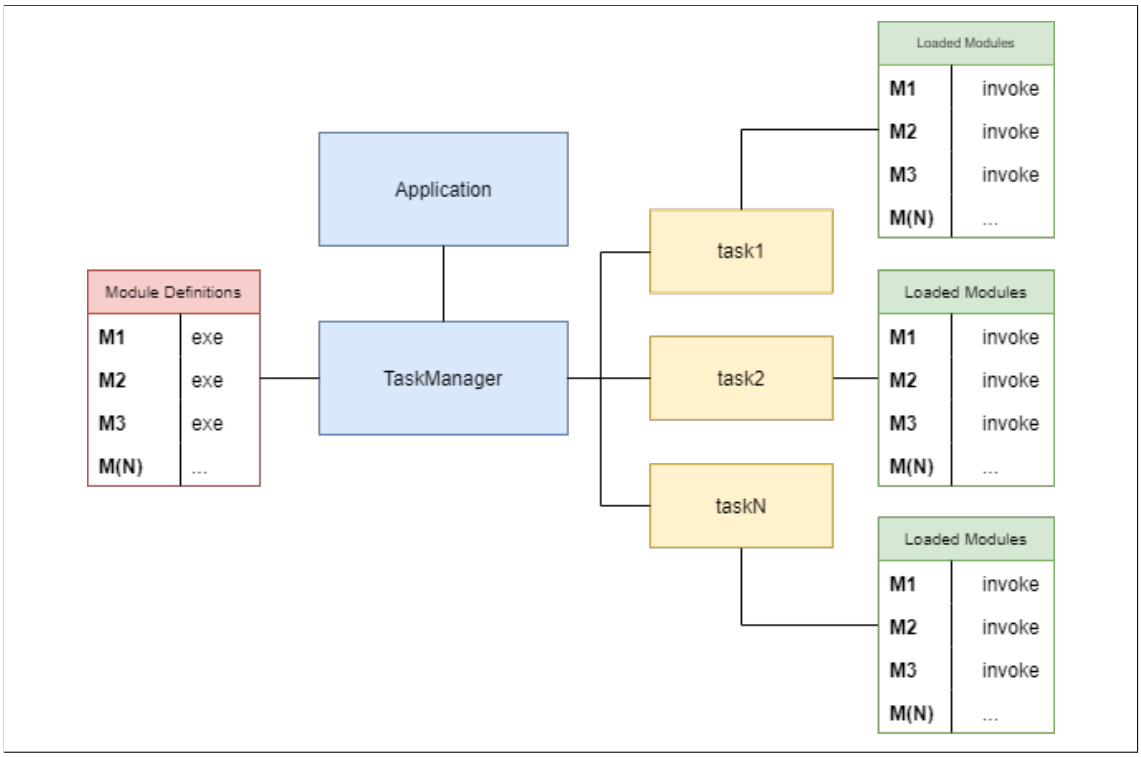


Figure 4.3: Loaded module representation

All modules adhere to the modular contract as shown in Figure 4.4 such that invoking a module returns a *Promise* [41] that resolves an *invoke* function. This function accepts a single parameter and returns a *Promise* that resolves the result of the module (which differs depending on whether a module performs extraction, transformation or loading operations). JavaScript *Promise* objects are state-representation of asynchronous operations in terms of success and failure of these operations. Since logic implemented in a module may not be asynchronous, all logic is wrapped within a `setImmediate` function to ensure that the contract of asynchronous execution of all modules is adhered to (except for where generators are used, since *ECMAScript 2017*® does not support asynchronous generator functions).

```
(function () {
    // Execution context ('this') is a config object
    function invoke(obj) {
        return new Promise((resolve, reject) => {
            setImmediate(() => {
                try { resolve(obj); }
                catch (error) { reject(error); };
            });
        });
    };
    return new Promise(async(resolve, reject) => {
        setImmediate(() => {
            try { resolve({ invoke: invoke }); }
            catch (error) { reject(error); };
        });
    });
})();
```

Figure 4.4: nETL module contract

On task execution (as directed by a user via the CLI) a task is run from beginning to end, iteratively extracting batches of data from a source, transforming that data, loading that data and then repeating the process. Since IO in JavaScript is asynchronous, batching either needs to be run sequentially (batches are processed one after the other), or by carefully managing asynchronous execution of batches. Batches extracted asynchronously and concurrently would quickly overwhelm the network capabilities of any computer since thousands of network requests would be queued instantaneously (network IO is many times slower than file IO, which is many times slower than data transformation). Most of these requests would fail - it is easier to serialize processing of batches than to queue network requests. As such, nETL is implemented to execute separate tasks concurrently; but a single task comprises a series of sequential steps.

JavaScript is not truly parallel - concurrent execution is achieved via adding procedures to an event loop that is executed on a first-in first-out basis via a single thread, with certain operations specified to be implemented asynchronously. Certain functions in JavaScript (`setTimout`, `setImmediate`, and `setInterval`) along with certain environment-provided APIs (such as the node.js filesystem API) pass control

back to the event loop before execution of these functions is completed. Such operations allow for specifying a callback function that is added to the event queue at time later from when the asyncrhonous function is first called. So if two tasks are running concurrently and a blocking procedure is called that is not asynchronous, processing of the event queue would be paused until the blocking procedure is completed. As a result, both tasks would be blocked until the procedure completes and the next item on the event loop is processed.

For this reason each step in the ETL engine (extraction, transformation and loading) is implemented asynchronously. Prior to the ECMAScript 2017 specification, an ETL engine implemented in node.js would have been challenging and require complex state management of asynchronous operations. But via the *async/await* API (a wrapper for JavaScript *Promises*), such state management is straightforward as shown in Figure 4.5 [2].

```
1  while (!batch.done) {
2      values = batch.value;
3      payload = await values.reduce(async(previousResults, item) => {
4          const results = await previousResults;
5          await asyncForEach(transformations, async(t) => {
6              item = await t.invoke.call(t, item);
7          });
8          if (item !== {} && item) results.push(item);
9          return results;
10     }, []);
11     loadResult = await load.invoke(payload);
12     batch = batches.next();
13 };
```

Figure 4.5: Loop with serialized asynchronous operations

4.2.1 Extracting data

Files are read in 64KB chunks from beginning to end within the context of an iterator created by a JavaScript *generator* function [40]. Chunks are held in memory, split

[2] Actually the extraction operation (`batch = batch.next()`) is NOT awaited (the `next` function is not asynchronous) because asynchronous generators are not supported in ECMAScript 2017

into lines (identified by *LF*, *CR* or *CRLF* line ending markers to allow for cross-platform portability) and yielded a single line at a time to a controlling function executed within the context of the iterative ETL engine. This function iteratively collects `n` lines at a time into a list (`n` is a user configurable property *batchSize*) and yields "lists of lines" - (*batches*). Generators are useful in the regard because they automatically create a state handling mechanism for iterating over file contents - i.e. pointers to positions in files, references to incomplete lines as retrieved from files, etc. Disk access via generators is achieved via code taken directly from an open-source library [13].

Lines in batches are then transformed concurrently, with transformations (specified in configuration) applied to each line in the batch in the order specified in the configuration. Concurrent processing of items accessed via a loop is achieved by wrapping the loop body in an asynchronous function (`setImmediate`), allowing the loop to progress without waiting for loop body execution to complete.

The loop itself is awaited, however, and once all transformation have been applied to all lines in the batch (lines can also be discarded from the batch depending on the transformations applied), the transformed batch is returned to `taskManager` and passed as an argument to the loading function specified via configuration. The function's contract is such that `taskManager` is notified when the batch has been loaded successfully to the destination, at which a further batch of lines is generated and processed. This batching loop is repeated until the extraction generator returns false when the end of a data source is reached.

4.2.2 Transforming data

In terms of processing lines in a flatfile (CSV format), headers are only ever read once with the assumption that the all rows can be split into values (by some defined delimiter) and that the order of the values corresponds with the order of the headers - if this is not the case, then the CSV is malformed. A reference to the CSV header row that is maintained for the duration of the transformation. CSV rows are iteratively

loaded into memory and split into values. Row values are matched with header values to form key:value pairs and create JavaScript objects

After transforming row-strings into row-objects, additional transformations are applied to each object. The following transformations are applied to objects:

- **Selection-filter:** Entire objects can be whitelisted based on properties and allowed values for those properties.
- **Join-selection-filter:** A list of attributes and values can be retrieved from a 3rd party data source (for example from a CouchDB index), and entire objects can be whitelisted based on the retrieved attributes and allowable values for those attributes. This is similar in concept to a join, although no means of actually joining documents is provided (i.e. creating attributes based on data retrieved from a 3rd party data source instead of applying selection - although this would be a fairly easy feature to implement).
- **Projection-filter:** Unneeded attributes can be removed from objects prior to loading into database/other destinations
- **Projection-append-attributes:** Additional attributes can be added to objects - e.g. a *type_* attribute can be added, along with a value as specified by configuration

4.2.3 Loading data

Batches of objects are serialized to JSON and are loaded into a CouchDB database via the the HTTP POST _bulk_docs endpoint (as opposed to separate network requests for each item in a batch). Bulk inserts are configured to be atomic - i.e. either an entire insert succeeds or fails. Network requests make use of the well-known, open-source node.js library "request" [48].

4.3 Setup

Running nETL requires an installation of node.js V8.9.0 +, which should include an installation of npm [44]. After cloning the nETL repository from Github to a local drive, dependencies should be restored using the npm CLI tool. Then the nETL app can be started from a terminal. Once the CLI is running, typing anything into the terminal and pressing enter outputs help where further direction can be obtained.

In conjunction with setting up nETL, a CouchDB server needs to be configured. This is easy to do on Windows machines - simply download the executable from apache.org and use the installer. Once installed the server should be run in single node configuration, binded to the 127.0.0.1 address. This allows access to the CouchDB UI via the browser at the address: `http://127.0.0.1:5984/_utils`, where first an admin user should be created. Working with databases via the CouchDB interface (called Fauxton) is straightforward.

Database creation involves only the single step of specifying a name and (optionally) security roles. CouchDB database configuration should be specified as part of creation - though this is only available when databases creation is specified via the HTTP interface and not the Fauxton GUI (the GUI doesn't allow for case-by case configration, but does allow for global configuration changes - although this is not recommended [8]); examples of configurable settings are sharding (q) and replica (n) count. For a single node setup, q=8 and n=1, meaning that a database has 8 shards and only 1 replica of each shard. This is the configuration used in this project. There is no point in storing more than one copy of a single shard on a single server, which is why n=1. For CouchDBs operation in cluster mode the default setup is q=8 and n=3. For clusters with a large number of nodes it might make sense to increase the value of the q parameter.

4.3.1 CouchDB Design Documents

Design documents are simply JSON strings, in which JavaScript functions are defined as strings. Working directly in JSON is unpleasant, however, and as such a open source tool, *couchapp*, written in Python is used for authoring and installing Design Documents to CouchDB databases [18]. This tool maps a directory structure to a JSON document; in other words, directory names become keys, and directory contents become values associated with these keys. Directories can be nested in the same way that JSON allows nesting objects; nested directories translate to nested keys in the final JSON document.

Using this tool, map and list functions can be authored as JavaScript files and don't have to be manually serialized to JSON. Working with map and list functions as JavaScript files allows for code completion, syntax highlighting and numerous other IDE-specific benefits. Isolated function files are also possible to unit test.

Chapter 5

Implementation

CouchDB's MapReduce implementation is limited in terms of performing joins and selections across multiple entities since indexed key:values pairs are derived from single documents. Each document is processed by MapReduce in isolation and the output is added to the index; no shared state between documents is available. In other words, map function executions are isolated both from each other and from the database. To have a shared state between documents and other documents/databases during MapReduce tasks would violate this principle, in addition to posing a significant security risk. By design, the MapReduce engine is pure JavaScript. No IO to either a file system or a network is possible. Although many JavaScript implementations provide APIs that allow this, such features are not part of the JavaScript specification and are not provided by couchjs.exe, the JavaScript MapReduce engine used by CouchDB [33].

Document selection can be performed during map function execution but requires selection predicates to be hard-coded into the map function; selections cannot make use of predicates that require joins as documents cannot be filtered based on values found in other documents. For the same reason, documents cannot be joined during map function execution. Instead joins can be accomplished during reduction, but this is not how reduction is intended to be used in CouchDB. Reduction is intended entirely for the purpose of value aggregation, and if aggregation is not performed then CouchDB's performance will deteriorate as index size increases.

Working with relational data in CouchDB therefore requires consideration across the entire software stack. That is, considerations with regards to ETL processes, indexing, and data retrieval should all be geared towards working with relational data. In this project the following process involving nETL and CouchDB is used:

- CSVs are parsed, rows filtered (selection), and data loaded into CouchDB by the nETL application
- An index is created from the CouchDB database via MapReduce
- Data are retrieved directly from the index file using a list function. This function performs joins and statistical calculations

MapReduce processes consist of separate functions defined for mapping documents from a database to an index (map functions), and aggregating values in the index (reduce functions). Only built-in reduce functions are used in this project since these are implemented within the main Erlang process and offer a performance benefit when compared to custom reduce functions. Unlike built-in reduce functions, custom reduce functions are executed externally to the main Erlang process by the couchjs.exe runtime and so an IO overhead is incurred by their usage. Running CouchDB on a Windows machine (as in this project) instead of Unix-based operating systems results in exaggerated overhead for custom reduce functions due to the differing IO implementation at the kernel level [32].

5.1 Joins

This project requires the student number fields in the three datasets to be joined; this field appears once for every student in the admissions data, several times for each student in the grades data and up to thousands of times per student in the events data. Both the grade and event data contain a field for year - i.e. the year in which the grade was obtained or the year in which an event was registered. Thus, for these two datasets it is necessary to join both the student number and year fields. Admissions data is only collected once per student, so it is associated with grades and events only by student number. Two joins are performed: a 2-way join of grades and admissions data (Equation 5.1), and a 3-way join of grades, events, and admissions (Equation 5.2).

$$grades \bowtie events \tag{5.1}$$

$$(grades \bowtie events) \rtimes admissions \tag{5.2}$$

5.1.1 Natural Joins

From a logical perspective, there are several ways a join can be achieved using CouchDB's MapReduce implementation. One such method is to configure a map function to output identical keys for the three entities, as described in the Equation 5.3:

$$[studentNumber, course, year] \tag{5.3}$$

During the map function's execution, it is possible to emit each document several times with different keys, which is useful when mapping documents that don't contain all the required fields of the key.

Grade documents contain fields for all three key values, so each grade document is emitted once. Admissions documents only contain one of the required keys (studentNumber), so the map function emits each admissions document several times - the document needs to be emitted for every possible courseCode and year that a grade document may have so that a natural join can be performed on a mutual [studentNumber,course,year] combination key. Similarly, events documents contain fields for studentNumber and year. Each events document needs to be emitted several times - once for each possible courseCode on which a natural join with grades documents may be required.

This approach to joining relies on grouping by common key (a natural join), which results in the reduce function receiving a list of all documents to be joined with each key. The join can then be performed in the reduce function (with difficulty considering that CouchDB's reduce function contract requires an allowance for `rereduce=true`).

To use a built-in reduce function to perform the join, entity output must be proxied by numeral values on which aggregations can be performed. This can be done via authoring a map function is to emit tuples as values. For example, to perform a

join on grade, admissions and events a map function can be configured to emit a tuple of 11 values corresponding information from different entities. To achieve this, on map function instantiation a tuple is instantiated with 11 values set to 0 (a falsy value). Each index (or group of indices) of the output tuple is reserved for entity-specific information and is adjusted if a document of the corresponding entity type is being processed. Values in the tuple associated with other entity types are left falsy (0), and so tuples are incorporated into indices on which aggregation will result in flattening many of these tuples into a single tuple containing information for all different entity types processed – in other words, a join is performed. In the context of this project, a join can be performed in this manner with the following tuple structure:

```
[
  0,                          # i = 0: a course % grade or 0
  0, 0, 0, 0, 0, 0, 0, 0,     # 0 < i < 8: admission grade %s
  0, 0                        # 8 < i > 11: event count for semester 1/2
]
```

If the document being processed by the map function is of *type_* 'grade', then the map function emits a tuple with a value at `i = 0` and 0s for all other indices:

$$[\%, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

If the document is of *type_* 'admission', the map function emits a tuple with values at `0 < i < 8`:

$$[0, \%, \%, \%, \%, \%, \%, \%, \%, 0, 0]$$

If the document is of *type_* 'event', the map function emits a tuple with values at `8 < i > 11`:

$$[0, 0, 0, 0, 0, 0, 0, 0, s1EventCount, s2EventCount]$$

In terms of performance this approach is disastrous. To analyze 40 courses taken over 3 years, each student's admissions document needs to be emitted 40 x 3 = 120 times so that the instance of the admissions entity always has a key that overlaps

with keys of instances of the grades entity. Likewise, each event document (which has a year but no course information) needs to be emitted 40 times - once for each course a join could potentially be performed on. This approach is wasteful of computer resources (if a single course is being analyzed) and completely impractical to scale (if several courses are being analyzed).

5.1.2 Joins via Sorting

A more performant approach to joining involves collecting adjacent values from indices sorted by keys designed specifically to place objects that need to be joined adjacently to each other. With reference to an index of the form shown in Figure 5.1 (A), documents output by a map function that share the same key are guaranteed to be grouped together since b+trees are guaranteed to be sorted. A map function outputting compound keys, such as in Equation 5.3, can fill in missing fields with 0. All three entities have a studentNumber field, so they will be grouped together. Admissions and events documents will be emitted with a value of 0 in place of the course field (so they will be ordered before the grades document output in the index). Additionally, admissions documents will be emitted with the value 0 in place of year, so they will always be ordered before events document output in the index.

Because output is systematically processed one student at a time, with grades, events, and admissions data processed in a predictable order for each student, joins can be achieved by holding documents for a student number in memory and processing grades, events, and admissions rows for that student number. When a new student number is encountered, a joined row is output, memory is flushed, and a new joined row for the student number is created. This approach is performant for 2-Way, 3-Way, and more generally, for N-Way joins. Additionally, since entity information is obtainable via key-structure - for example a key of $(studentNumber, 0, 0)$ is indicative of the admissions entity - map output can be structured according to entity types and map function output doesn't have to be proxied for the potential of processing every type of entity.

5.1.3 Joining and Aggregating

CouchDB's reduce functions are primarily geared towards aggregating data. This is particularly useful for the events data where several thousand events documents are associated with a single student. The documents are grouped when passed to the reduce function and aggregated to a single output in the final index (for example, by specifying the _stats or _sum reduce functions).

In other words, only a single document that is an aggregation of all events documents will be stored as reduced output in the view. So, when using reduction for any student number, scanning the index first produces an admissions document, then a single (aggregated) events document, then a single grade document for each course that the student enrolled in. An example of this with reference to the _sum reduce function is shown in 5.1 (B). Retrieving reduced index output requires querying the index and specifying `reduce=true`. In addition, it is necessary to specify `group=true`. This results in the reduction being performed on grouped keys instead of retrieving an aggregation of the entire index (which can be useful, for example, when calculating variance).

```
1  /* (A) reduce = false */
2  [<ID>, ‘0’, 1]: [0, b1, b2, b3, b4, b5, b6, b7, b8, 0, 0]
3  [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
4  [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
5  [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
6  [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
7  [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
8  [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9  [<ID>, ‘CSC1015F’, <Year>]: [98, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
10 [<ID>, ‘MAM100F, <Year>]: [94, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
11
12 /* (B) reduce = true & group = true */
13 [<ID>, ‘0’, 1]: [0, b1, b2, b3, b4, b5, b6, b7, b8, 0, 0]
14 [<ID>, ‘0’, <Year>]: [0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3]
15 [<ID>, ‘CSC1015F’, <Year>]: [98, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
16 [<ID>, ‘MAM100F, <Year>]: [94, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 5.1: Index structure

With this approach, MapReduce fills the important role of structuring aggregated

data as a sorted b+tree index that then facilitates joining. But it should be noted that joining is possible because indices are structured as b+trees - the same structure that is used by the main database files. These files are also sorted (according to the document's _id field). As such, it is possible to join documents directly upon retrieval from the main database rather than creating an index first - especially since the _id field can be given meaningful values. There are benefits to indexing database files, however, such as those listed here:

- CouchDB database files are sorted by the "_id" field, which, when unspecified on the document insert, is initialized as a UUID. Using UUIDs as unique document identifiers allows for distributed systems in which cluster nodes can operate independently of each other without the possibility of documents being created in separate nodes with conflicting IDs. Even though not required for this project, such best practices have been followed. A b+tree sorted by a UUID is not useful for document retrieval, and as such, views are required of the underlying data store for any kind of index-based querying
- List functions are invoked via an HTTP GET request with the requirement of specifying a view within the URI. List functions are convenient for usage in this project as they facilitate ordered, iterative, range-based access (meaning that data can be accessed sequentially, in isolation and in reliable order)
- When aggregations of specific entities are required, retrieving data directly from these indexes is vastly easier than having to aggregate during data retrieval. Without a reduce function, additional logic would be required during retrieval to perform such aggregations. As aggregation logic becomes more complicated, the difficulty of such direct data retrieval increases and the benefit of using indexes increases as a result. Also, performing aggregation during the indexing phase instead of during data retrieval greatly improves the performance of data retrieval

5.2 Selections

Performing joins by iterating over ordered indices is quite efficient in terms of memory usage. No matter the size of the datasets being processed, memory usage will always be fairly low. An increase in the size of datasets will simply result in more processing time. It is therefore possible to perform selections during index calculation via a map function. To do this, predicates are hard coded into the map function and applied to every document that is processed. Each processed document is then either emitted and incorporated into an index or discarded.

The limitation of this approach is that the predicates can only be applied to fields of the documents being processed. It is a common use case to apply selection predicates to fields made available through first joining a dataset with another dataset. This is impossible to achieve within the context of a map function's execution. It is possible to apply selections that require joining data on index retrieval (as a join is performed), but this is quite inefficient in terms of the size of the required index. The events data, for example, has 44.4 million documents, most of which are not required. To perform a selection on the events data during MapReduce requires iterating through all events documents - which is expensive in terms of time. Reducing the number of documents loaded in the first place makes working with CouchDB easier and more performant. As such, where a join is required to make available fields used in a selection predicate, a join is (effectively) achieved used nETL during the ETL phase of analysis. Joins can therefore be performed by the nETL application is two ways:

- Selection-based predicates can be applied to fields based on field values. More advanced predicate logic (i.e. where field values contain substrings, or where field values fall in a range, etc. - i.e. joins based on conditions other than equality) is not implemented in the current version of nETL that was used for this project, although it would be fairly straightforward to implement such a feature in the future. Such predicates are not used in this project, but if they were, they could easily be implemented in the map function (provided a join

isn't first required)

- Selection-based predicates based on information retrieved from a separate CouchDB index (similar in concept to a join). Such an approach is conceptually similar to performing joins on distributed datasets via a *semi-join* where a list of keys requiring data joins is acquired prior to performing database operations, thus minimizing network transport costs [12].

When processing admissions and events CSVs, a join with the grades entities is required to make a course code field available for only those student numbers that are associated with the CSC1015F course. To allow for this, a database is set up housing all CSC1015F grades documents. Indices are created for this database (a database of grades) to make a list of students available in whatever format is required: for admissions data, a predicate is applied based on the *anonIDnew* field; for events data, a predicate is applied based on the *uct_id* field. Indices created from the grades database provide a list of student numbers for these different field names (the list of student numbers is the same).

5.3 Statistical calculations

Because all numbers in JavaScript are 64-bit floating-point (following the IEEE 754 standard [28]), working with rational numbers with decimal points results in peculiarities when compared to a base-10 number system - for example, the sum: $0.1+0.2 = 0.30000000000000004$. Decimals such as 0.1 and 0.2 cannot be accurately represented in binary format within a 64-bit address space (or any finite amount of memory). As such, rounding errors occur. Quantifying the margin for such errors and handling these cases correctly is difficult [27], so to side-step this uncertainty an open-source library (*decimal.js* [38]) is used to handle arithmetic in JavaScript. CouchDB allows for the usage of 3rd party JavaScript libraries through the implementation of the CommonJS specification within the context of couchjs.exe [1].

Numerical data is treated statistically during index retrieval, and worked out using the Decimal.js library and well-known statistical formulae, as shown in Equations 5.4

(variance), 5.5 (computational re-arrangement of the variance formula), 5.6 (standard deviation), and 5.7 (correlation).

$$(\sigma_{\overline{x}})^2 = \frac{\sum (x - \bar{x})^2}{n - 1} \tag{5.4}$$

$$(\sigma_{\overline{x}})^2 = \frac{\sum x^2 - \frac{(\sum x)^2}{n}}{n - 1} \tag{5.5}$$

$$\sigma_{\overline{x}} = \sqrt{(\sigma_{\overline{x}})^2} \tag{5.6}$$

$$r = \frac{N \sum xy - (\sum x)(\sum y)}{\sqrt{[N \sum x^2 - (\sum x)^2][N \sum y^2 - (\sum y)^2]}} \tag{5.7}$$

5.4 Aggregation Example

Admissions data can be used to profile students based on their Grade 12 results, NBT scores, or some combination of the two. In other words, several benchmarks datasets can be derived from the admissions data. The variance and standard deviation of each of these datasets is calculated below to provide an example of aggregation using CouchDB.

5.4.1 ETL

Rows are extracted from the admissions data in batches of 5 000. Each row is converted into an object, and then the objects for undergraduate students who have citizenship or permanent residency in South Africa and who have a grade for the CSC1015F course are selected. A *type_* attribute with the value 'admission' is added to each row (now an object). Batches of objects (there are at most 5 000 objects per batch, but usually far fewer due to the filtering) are loaded into a CouchDB database. An example of a row from the admissions data serialized to a JSON

string and loaded into CouchDB is shown in Figure 5.2.

```
{
    "_id": "6587aa5b36bba2a70aeba96d06f05d2b",
    "_rev": "1-8c7d395e046e452442907e3388c74b41",
    "anonIDnew": 3103212,
    "Eng Grd12 Fin Rslt": 58,
    "Math Grd12 Fin Rslt": 73,
    "Mth Lit Grd12 Fin Rslt": "",
    "Adv Mth Grd12 Fin Rslt": "",
    "Phy Sci Grd12 Fin Rslt": 67,
    "NBT AL Score": 53,
    "NBT QL Score": 43,
    "NBT Math Score": 60,
    "RegAcadYear": 2016,
    "type_": "demographic"
}
```

Figure 5.2: Serialized admissions document

5.4.2 Indexing

All the admissions documents are loaded into the CouchDB database and mapped to an index consisting of *anonIDnew:[benchmarks]* key-value pairs. Derivative benchmarks are calculated during map function execution before being incorporated into the index – each value added to the index is a tuple of benchmarks categorized according to the following indices:

```
Gr12 Eng %
Gr12 Sci %
Gr12 Mth %
NBT AL %
NBT QL %
NBT Mth %
Avg Gr12 %
Avg Gr12 % (Dbl Mth)
Avg Gr12 % (Dbl Mth \& Sci)
Avg NBT %
Avg NBT % (Dbl AL)
Avg NBT % (Dbl QL)
Avg NBT % (Dbl Mth)
Avg NBT % (Dbl AL/QL)
Avg NBT % (Dbl AL/Mth)
```

```
15  Avg NBT % (Dbl QL/Mth)
16  Avg Gr12 & NBT
17  Avg Gr12 & NBT (Dbl Gr12 Mth)
18  Avg Gr12 & NBT (Dbl Gr12 Mth & Sci)
```

The built-in _stats function is used to aggregate values in the view-index by benchmark type, which when used with `group=false`[1] results in the index output of a single stats-object, as shown in Figure 5.3[2] (indices of each object in the value output are indicated on the right-hand side of the figure). The objects produced by the _stats function (one object per benchmark dataset) contain fields for:

- The sum of all the items in the dataset
- The count of all the items in the dataset
- The smallest number in the dataset
- The largest number in the dataset
- The sum of each item squared in the dataset

[1] index keys are ignored and treated as `null`, as a result all index values are grouped together
[2] Values are rounded for better display

```
{"rows": [{
"key": null,
"value": [
  {"sum": 71751,"count": 908,"min": 50,"max": 97,"sumsqr": 5720599},  // 0
  {"sum": 74174,"count": 908,"min": 48,"max": 100,"sumsqr": 6151326}, // 1
  {"sum": 78802,"count": 908,"min": 63,"max": 100,"sumsqr": 6900682}, // 2
  {"sum": 67207,"count": 908,"min": 33,"max": 94,"sumsqr": 5057191},  // 3
  {"sum": 69713,"count": 908,"min": 27,"max": 98,"sumsqr": 5512393},  // 4
  {"sum": 69136,"count": 908,"min": 29,"max": 98,"sumsqr": 5439872},  // 5
  {"sum": 74909,"count": 908,"min": 61,"max": 98,"sumsqr": 6227518},  // 6
  {"sum": 75882,"count": 908,"min": 62,"max": 98,"sumsqr": 6389866},  // 7
  {"sum": 75541,"count": 908,"min": 61,"max": 98,"sumsqr": 6338061},  // 8
  {"sum": 68685,"count": 908,"min": 39,"max": 94,"sumsqr": 5288410},  // 9
  {"sum": 68316,"count": 908,"min": 40,"max": 94,"sumsqr": 5221243},  // 10
  {"sum": 68942,"count": 908,"min": 36,"max": 94,"sumsqr": 5337861},  // 11
  {"sum": 68798,"count": 908,"min": 40,"max": 95,"sumsqr": 5315192},  // 12
  {"sum": 68595,"count": 908,"min": 39,"max": 94,"sumsqr": 5272344},  // 13
  {"sum": 68412,"count": 908,"min": 40,"max": 94,"sumsqr": 5238199},  // 14
  {"sum": 68981,"count": 908,"min": 37,"max": 95,"sumsqr": 5346677},  // 15
  {"sum": 71797,"count": 908,"min": 52,"max": 94,"sumsqr": 5730957},  // 16
  {"sum": 74132,"count": 908,"min": 56,"max": 96,"sumsqr": 6102577},  // 17
  {"sum": 74142,"count": 908,"min": 56,"max": 97,"sumsqr": 6107676}   // 18
]
}]}
```

Figure 5.3: Aggregated output from *_stats*-function

5.4.3 Presentation

With reference to the reduced (aggregated benchmarks) output shown in Figure 5.3, variance and standard deviation are worked out according to Equation 5.8, with standard deviation being the square root of variance. The computational variation of the sample variance formula is used.

$$(\sigma_{\overline{x}})^2 = \frac{sumsqr - \frac{sum^2}{count}}{count - 1} \tag{5.8}$$

Variance and standard deviation are calculated directly from the reduced index output during data retrieval using a CouchDB list function, with the results output in tabular form, as shown in Table 6.2 (page 71).

5.5 Example of a 2-Way Join

Correlating features across separate datasets requires first joining these datasets. Correlation coefficients (r) between different admissions benchmarking methods and CSC1015F course results are calculated according to the inner join shown in Equation 5.1. A single student number may appear multiple times in the grades data if a student repeated a course, but should appear only once on the admissions data, for the first time they registered at UCT. Rows for student numbers found in the admissions data but not the grades data are not used, nor are rows found for students in the grades data but not the admissions data.

5.5.1 ETL

Using nETL, rows are extracted from the two CSV files (*Admissions (2014 - 2016).csv* and *Grades (2014 - 2016).csv*) concurrently and independently of each other, in batches of 5 000 and 10 000 rows respectively.

Through nETL configuration, rows from the admissions data are selected for students who are South African citizens or permanents residents, and who are undergraduates. Rows from the grades data are selected for students who attended CSC1015F during 2014, 2015, or 2016. Since the admissions data doesn't have a field for course year, a natural join with grades on the *anonIDnew* field is performed via nETL ($admissions \bowtie grades$) to retrieve a list of students who attended CSC1015F. Using this list, rows are selected from the admissions data for students who attended CSC1015F.

Rows retrieved from the CSVs contain numerous fields that are not required, and so nETL is configured to apply an attribute-whitelisting process to both admissions and grades data. Batches of objects are serialized to JSON strings and loaded into a single CouchDB database using the *_bulk_docs* endpoint. An example of a row from grades data serialized to a JSON string and loaded into CouchDB is shown in Figure 5.4.

```
{
    "_id": "7530f4eed7e6bc3ef0d99a53be8ba9a2",
    "_rev": "8-232d0cf39728d41b4c5935f12469209d",
    "RegAcadYear": 2016,
    "anonIDnew": 1,
    "Course": "CSC1015F",
    "Percent": "55",
    "type_": "grade"
}
```

Figure 5.4: Serialized grades document

5.5.2 Indexing

After loading the data from the CSVs into CouchDB, a map function is used to produce an index of the CouchDB documents ordered by Student ID, with the guarantee that for everystudent number, documents are ordered by type; an admissions document output precedes grade documents output for any given student. Knowing the order of documents through the view-index allows the join to be performed upon data-retrieval. Only a map function is required (no reduce function). That is, on the map function's execution the "type_" attribute is checked. If the document is a line of the grades entity, then the key [Student ID, Course, year] is emitted along with a single number for the value - the percent achieved for the course. If the document is a line of the admissions entity, then the key [studentNumber, 0, 0] is emitted along with an ordered list of 19 values corresponding to each of the 19 different methods of benchmarking students (discussed in Chapter 5.4). A key of [studentNumber, year] could have been used instead, since the course is always CSC1015F. But explicitly including the year in the key makes debugging easier, whilst also making the code more generically applicable if other courses are to be analyzed.

Normalization of the percentage fields (i.e. "Percent" for the grades entity and the test results in the admissions entity) is done using a nested function (a function defined within the map function) according to best-guess logic on how grade symbols correlate with percentage.

Because a student should only be represented by a single row in the admissions

data and should only achieve a single grade per course per year, this 2-way join is achievable without using a reduce function. There is also no need to aggregate rows from either the admissions or grades data prior to joining. The logic of the map function is shown in the activity diagram in Figure 5.5.

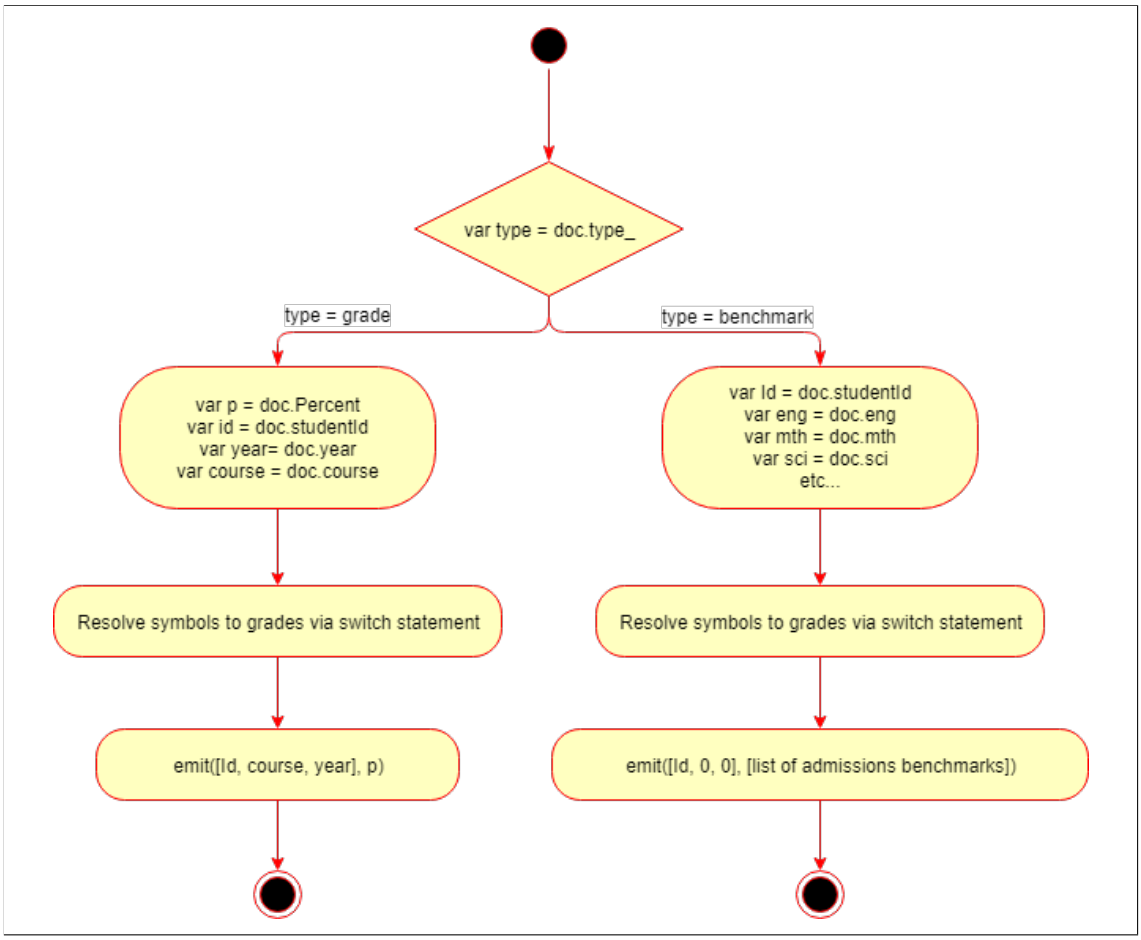


Figure 5.5: *Map*-function: grades ⋈ admissions

5.5.3 Presentation

A list function is used for data retrieval; this function scans the index iteratively (i.e. documents for each student are processed iteratively; first a student's admissions document is processed, then a student's grades documents are processed). The 2-way join is achieved in this way; for every student number the grade and admissions data are joined, and summations of various fields of the grades and each benchmark are updated. For example, for every index key:value processed, the product of

`CSC1015F \% x each benchmark \%` is calculated and added to running sums kept for each benchmark.

Once the iteration over student numbers is finished, the completed summations are used to calculate correlation coefficients for each combination of benchmarking method and grade according to Equation 5.7[3].

Although the _stats reduce function calculates *sum of squares* per dataset, this is not useful in cases where individual rows from separate entities should be joined (such as in this 2-way join). For example, the numerator (Equation 5.9) from the correlation formula (Equation 5.7) cannot be calculated during MapReduce. Only the $\sum x$ and $\sum y$ values are accessible when joining the two entities during list function execution, and not x and y values since these values come from entity instances.

$$N \sum xy - (\sum x)(\sum y) \tag{5.9}$$

Only aggregations of entity instances are available upon index retrieval (when retrieving reduced output) and not the individual instances. Similarly, the denominator of the formula also could not be calculated from the _stats function output.

List function logic is represented as an activity diagram in Figure 5.6 and is configured to output a table of correlation coefficients for each benchmarking method as shown in Table 6.3 (page 72).

[3] x: grade %, y: benchmark (r is calculated for multiple y values)

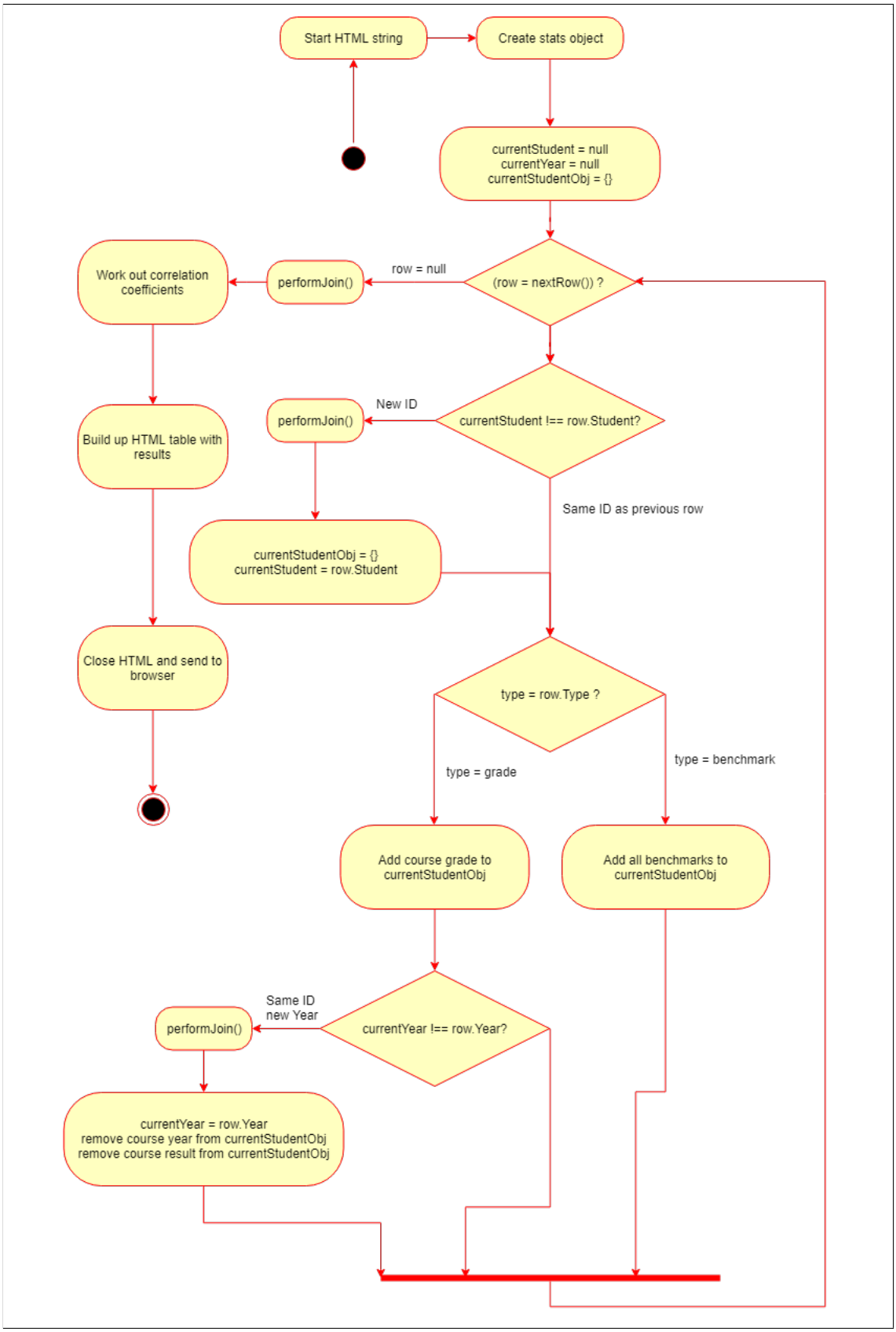


Figure 5.6: *List*-function: grades ⋈ admissions

5.6 Example of a 3-Way Join with Aggregation

By performing a 3-Way join alongside aggregation, the correlation between CSC1015F course grades and Sakai usage is tested by working out the change of each student's class rank according to each of the benchmarks compared to their CSC1015F grade class ranking. Each benchmark/grade ranking change is then compared to Sakai usage. To achieve this, for each benchmarking dataset a derived dataset is created that contains numerical values of changes in class rank. Correlation between these datasets and Sakai usage is tested by finding correlation coefficients for each derived classRankChange dataset compared to the Sakai usage dataset. For this project, Sakai usage is proxied in terms of the number of times a student logs into the system - the rationale being that the more often students log into Sakai, the more they are using it.

5.6.1 ETL

Using nETL, rows are extracted from the three CSV files (*Admissions (2014 - 2016).csv*, *Grades (2014 - 2016).csv* and *Events (2016).csv*) concurrently and independently of each other, in batches of 5 000, 10 000 and 30 000 rows, respectively.

By using nETL configuration, rows from the admissions data are selected for students who are South African citizens or permanents residents, and who are undergraduates; rows from the grades data are selected for undergraduate students who attended CSC1015F during 2016; rows from the events data are selected for presence events only. Dynamic filters are configured for the admissions and events data to only include students who took the CSC1015F course.

The events data contains a field *ref*, which is a long string and is not required in the analysis. As such, this string is dropped through a whitelisting process prior to serialization and by loading strings into CouchDB (in batches via the *_bulk_docs* endpoint). An example of a row from the event data serialized to a JSON string is shown in Figure 5.7.

```
{
    "_id": "000e569ee321b915bae59fe62e0051e3",
    "_rev": "1-7112afce121087818c33ebfd0fd7fed7",
    "event_date": "2016-04-17T14:04:20.000Z",
    "event_id": 281, // anonymized student number
    "uct_id": 3018438,
    "site_key": 2297,
    "type_": "event"
}
```

Figure 5.7: Serialized events document

5.6.2 Indexing

Since a single student may be associated with many rows in the events data (sometimes even thousands of rows), a reduce function is used within the MapReduce job to aggregate the events rows into a single document, which provides a count of Sakai presence events for both the first and second semesters, with the output of this aggregation included in the index along with grade and benchmark data when reduce = true. The index consists of key:value pairs of student numbers associated with a tuple that contains values for grades, benchmarks, and event information.

During the map function's execution, the logical handling of grade and admissions entities has already been discussed. If the document produced is a line from the events entity, then the date of the event is categorized as either having occurred in semester 1 (S1) or semester 2 (S2). A key of [Student ID, 0, Year] is emitted along with the tuple [S1, S2]. The S1 and S2 variables are 0 by default, and depending on the date of the presence event, one of these variables is altered to '1'. CSC1015F is a first-semester course, but by including a count of both semesters, it becomes possible to use the same indexing code on courses that run in the second semester as well. The logic of the map function is shown in Figure 5.8.

Using the _sum reduce function, an aggregation is done across all documents with the same key; this means that for each student, an aggregation is performed on a single grades document, a single admissions document, and many events documents in which the S1 and S2 variables are summed to form the tuple [sum of S1, sum

of S2]. The key emitted for each type of entity is designed so that the view-index is ordered by StudentID. For each student number, documents are ordered by the second key (course), which means that admissions and events entities are sorted before the student's grades; and the 3rd component of each key results in admissions data that always precedes events documents. As such, during view-index retrieval it can be taken as a given that for a single student ID, benchmark values will be retrieved first, followed by event values, followed by grade values.

5.6.3 Presentation

A list function is implemented to retrieve the index with `reduce = true \& group = true` so that only reduced index output is retrieved, grouped by key.

Because the ranking of students for course grade and each benchmarking method is required, several ordered lists are kept in memory for the duration of time that a single course code is being processed (in this case CSC1015F). In other words, for each course code, and then for every result retrieved from the index, the student number from the row being processed is checked and compared to the student number from the previous row. If the current student number is not the same as the previous student number, then the current rows in memory are joined, and list of ranks for each course and benchmark are updated. It is necessary to process all index output for a particular course code before students can be ranked for a course (as well as ranked in terms of their benchmarks, when compared to other students who took that course).

As joins are performed, an object is kept in memory for each student, which keeps track of their course grade and scores for each benchmark. Once a single course's results have been iterated over, ranking lists are created that comprise tuples of [StudentId, %] and are ordered by the second index ($i = 1$). The object of students is updated to add a rankChange value for every benchmarking method compared to the course grade. This is worked out as $courseRank - benchmarkRank$.

Then, by iterating over the student object, it is possible to perform the summations

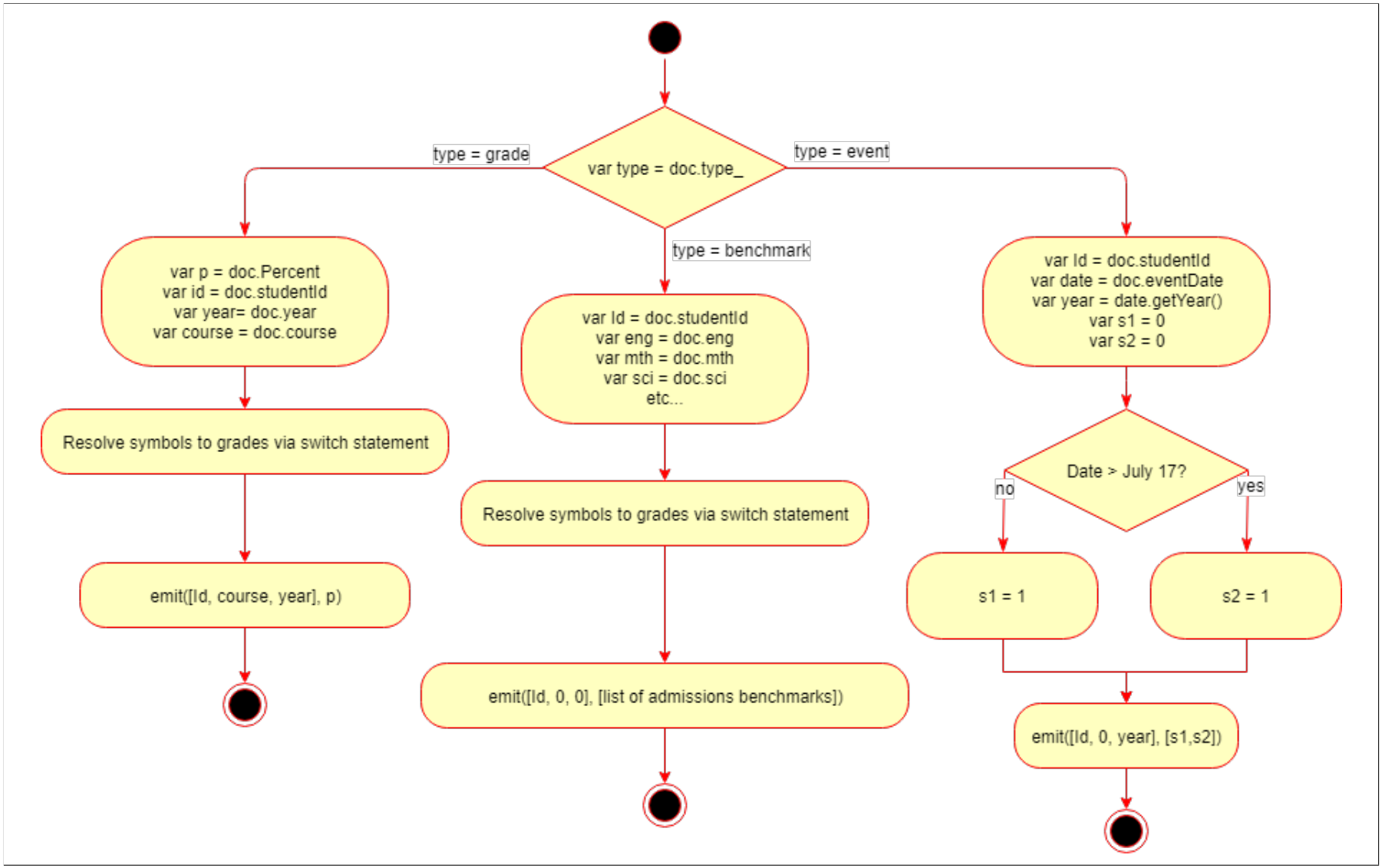
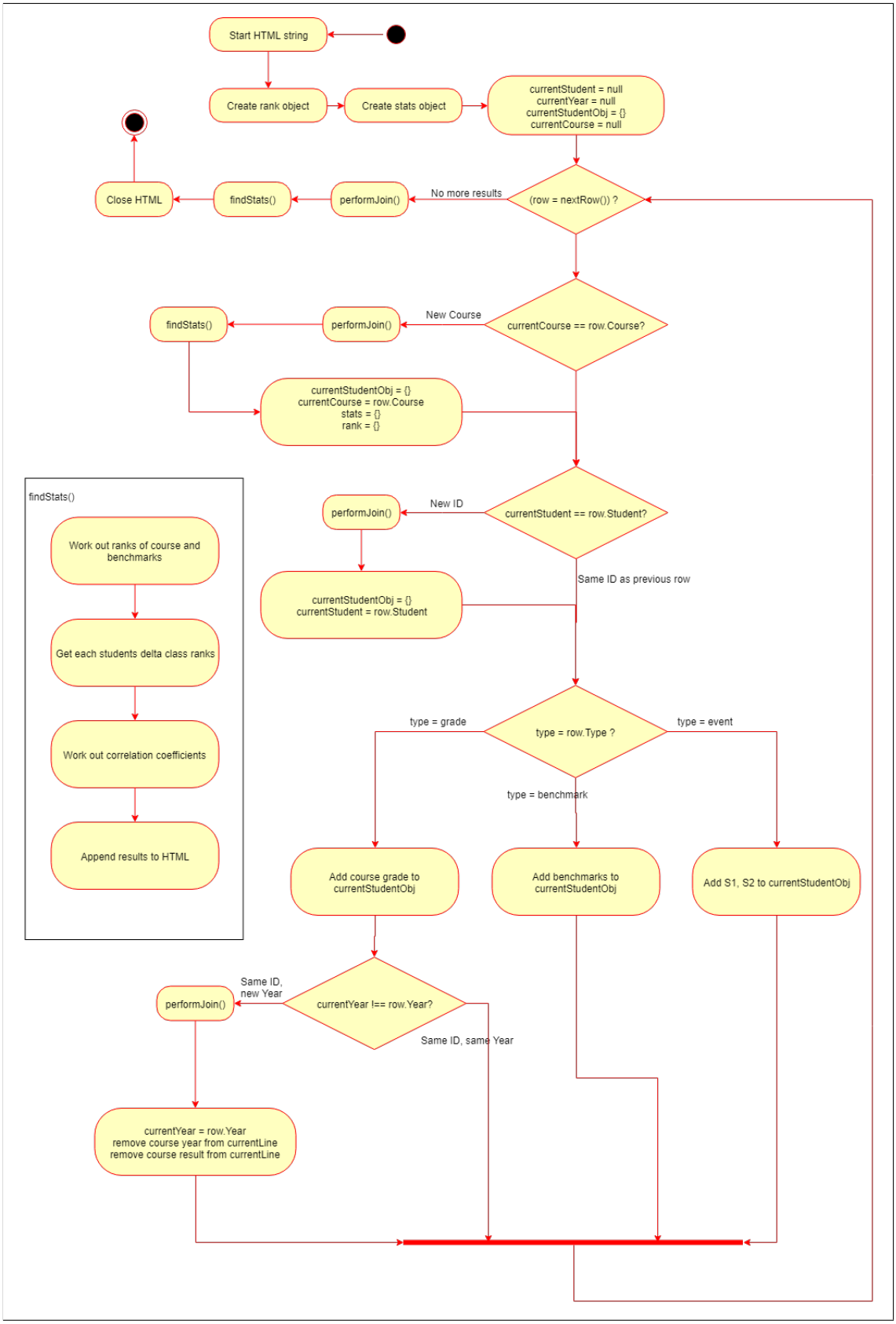


Figure 5.8: *Map*-function: (grades $\bowtie$ events) $\rtimes$ admissions

stipulated in Equation 5.7, and the correlation between change in class rank and events count is found (a separate correlation corresponding to each benchmark is obtained).

List function logic is represented as an activity diagram in Figure 5.9 and is configured to output a table of correlation coefficients for each benchmarking method as shown in Table 6.4 (page 73).



***List*-function: (grades ⋈ events) ⟖ admissions**

Chapter 6

Results

This chapter discusses usage and testing of the system, and summarizes the results obtained from the case study.

6.1 Systems Summary

Measurable aspects of the systems used during the course of this project such as: runtimes of nETL processes, indexing time in context of the data size, and database storage size are included in Table 6.1. Indexing time for a CouchDB database of 3.9 million documents is about 140 seconds.

Task runtime is quicker executed in isolation than concurrently with other tasks, which is expected; task runtime increases when interleaved with other tasks since nETL tasks are executed concurrently but not in parallel. Multiple task executions taking similar amounts of time to complete, and performance seems tolerable, taking around and hour to process a CSV of 44.4 million lines.

Table 6.1

Summary of nETL and indexing runtime, and data-selection volumes for aggregation example, 2-way join and 3-way join analyses

	Aggregation	2-Way Join	3-Way Join
Admissions lines extracted[*]	12 219	12 219	12 219
Admissions lines loaded[**]	1 381	1 381	595
Admissions task time (sec)	1.588[1a]	2.712[1b]	4.935[1c]
Grades lines extracted		513 872	513 872
Grades lines loaded		1 891	738
Grades task time (sec)		50.993[2a]	97.532[2b]
Events lines extracted			44 420 508
Events lines loaded			661 555
Events task time (sec)			3 875.932[3]
Database size after load (MB)	< 1	< 2	169.5
Indexing time (sec)	0.533[4a]	1.092[4b]	140.763[4c]

[*] *lines extracted* refers to how many rows of data a CSV contains

[**] *lines loaded* refers to selected number of lines from CSVs

[1a] Average of 3 runs: $1.502, 1.745, 1.518$

[1b] Average of 3 runs: $3.045, 2.787, 2.305$

[1c] Average of 3 runs: $5.313, 3.548, 5.782, 5.095$

[2a] Average of 4 runs: $52.899, 51.399, 48.681$

[2b] Average of 4 runs: $94.353, 92.902, 100.134, 102.738$

[3] Average of 3 runs: $3989.82, 3942.93, 3695.046$

[4a] Average of 3 runs: $0.519, 0.472, 0.607$

[4b] Average of 3 runs: $0.988, 1.347, 0.941$

[4c] Average of 3 runs: $141.25, 142.709, 138.331$

6.2 Tests

Accurate nETL application component functionality is confirmed via unit testing, as are map and list functions used for index creation and retrieval. Unit tests are written using the open source JavaScript *Mocha* testing framework [39]. In addition, a small sample of the data was processed manually to confirm that the processes

work as expected and that statistical calculations produce results as expected.

6.2.1 nETL Unit Tests

The basic premise of the ETL process is that the lines are extracted from CSVs and loaded into CouchDB reliably. Assertions are used to ensure that each nETL module (the extraction module, the transformation modules, and the loading module) perform as expected. No integration tests are performed except by manually checking that the test data is all loaded into CouchDB and in the anticipated format - which is indeed the case.

Tests relating to CSV-line extraction assert that all lines from the CSV are extracted iteratively and not all loaded into memory at once, and that all lines are extracted from CSVs. Tests for parsing CSVs into objects ensure that CSVs are treated according to the RFC 4180 specification; qualifiers are handled correctly, columns line up correctly with the headers, values are handled correctly, and lines are correctly transformed into JavaScript objects. In terms of selection, unit tests ensure that objects may be filtered for individual values for up to multiple attributes, that objects can be filtered on any number of attributes, and that filtering is done on an all-or-nothing-basis (objects either meet all filter requirements or are returned as "null"). Unit tests assert that creation and whitelisting of attributes works as expected.

6.2.2 Manual Map & List Function Tests

Manually testing each of the processes described in Chapter 5 was done using small dummy datasets with just 5 IDs. Based on this dataset, the MapReduce output of test data is as expected:

- Correlation between Grades and admissions benchmarks
  - Document output is ordered key[0], then[1], then key[2] - For every student output of benchmark data is followed by that student's grade data

  - Multiple results from the same course are output in order, ordered by course registration date.
  - Value output for the Grade documents is a single number, and output for the Benchmark document is an array of 8 numbers (that correspond to the CSV input)
  - No documents appear that should be filtered out
  - Output contains the correct number of documents
- Correlation between Sakai presence events and $\Delta ClassRank$
  - Event counts are correct, and the output format is correct for semester 1 and semester 2 for each student
  - Document ordering is correct for each student (benchmarks output, followed by events output, followed by *grades* output)
  - Grade documents from years other than 2016 are not included in the output

List function output of the test data shows that rows are joined correctly for both correlation analysis of grades (A) and event/$\Delta classrank$ (B). One particular case worth testing is that when MapReduce output includes benchmarks and event counts, but no grade documents for a student, the List output does not include that student. This case often occurs when a student's took CSC1015F in a year other than 2016, and that students event data includes usage for courses other than CSC1015F in 2016. Statistical calcluations were checked for accuracy using Microsoft Excel (when possible).

6.3 Student Profiling

Exploration of datasets as part of the data mining process includes assessing properties such as variance and correlations within the data. These impact the usefulness of attributes or features in terms of data mining and so are important to quantify during the data exploration phase.

Table 6.2 lists variance measurements for several datasets derived from the admis-

sions data in an effort to assess different means of benchmarking students. These same datasets are tested for correlation with final CSC1015F grade results as shown in Table 6.3. An additional dataset is compiled from each benchmarks dataset comprising students change in rank from CSC1015F course grades compared to their class rank in the benchmark dataset. Correlation between this dataset and the events dataset is shown in Table 6.4.

Table 6.2

Variance and Std. Deviation of different possible metrics for benchmarking students during admissions

Benchmark	$(\sigma_{\overline{x}})^2$	$\sigma_{\overline{x}}$
Gr12 Eng %	56	7.5
Gr12 Sci %	101.5	10.1
Gr12 Mth %	68.1	8.3
NBT AL %	91.2	9.6
NBT QL %	176.5	13.3
NBT Mth %	193.8	13.9
Avg Gr12 %	52.5	7.2
Avg Gr12 % (Dbl Mth)	53.3	7.3
Avg Gr12 % (Dbl Mth & Sci)	59	7.7
Avg NBT %	102.3	10.1
Avg NBT % (Dbl AL)	89.7	9.5
Avg NBT % (Dbl QL)	113.8	10.7
Avg NBT % (Dbl Mth)	113	10.6
Avg NBT % (Dbl AL/QL)	99.6	10
Avg NBT % (Dbl AL/Mth)	92.3	9.6
Avg NBT % (Dbl QL/Mth)	117.1	10.8
Avg Gr12 & NBT	59.4	7.7
Avg Gr12 & NBT (Dbl Gr12 Mth)	55.3	7.4
Avg Gr12 & NBT (Dbl Gr12 Mth & Sci)	59.1	7.7

Table 6.3

Correlation between different benchmarking methods and CSC1015F grades

Benchmark	r
Gr12 Eng %	0.287
Gr12 Sci %	0.465
Gr12 Mth %	0.447
NBT AL %	0.368
NBT QL %	0.533
NBT Mth %	0.510
Avg Gr12 %	0.485
Avg Gr12 % (Dbl Mth)	0.487
Avg Gr12 % (Dbl Mth & Sci)	0.493
Avg NBT %	0.583
Avg NBT % (Dbl AL)	0.559
Avg NBT % (Dbl QL)	0.580
Avg NBT % (Dbl Mth)	0.583
Avg NBT % (Dbl AL/QL)	0.567
Avg NBT % (Dbl AL/Mth)	0.570
Avg NBT % (Dbl QL/Mth)	0.589
Avg Gr12 & NBT	0.610
Avg Gr12 & NBT (Dbl Gr12 Mth)	0.587
Avg Gr12 & NBT (Dbl Gr12 Mth & Sci)	0.578

Table 6.4

Correlation Sakai presence and (course class rank - benchmark class rank)

Benchmark	r
Gr12 Eng %	0.007
Gr12 Sci %	-0.091
Gr12 Mth %	-0.038
NBT AL %	0.144
NBT QL %	0.166
NBT Mth %	0.017
Avg Gr12 %	-0.073
Avg Gr12 % (Dbl Mth)	-0.070
Avg Gr12 % (Dbl Mth & Sci)	-0.076
Avg NBT %	0.119
Avg NBT % (Dbl AL)	0.128
Avg NBT % (Dbl QL)	0.138
Avg NBT % (Dbl Mth)	0.087
Avg NBT % (Dbl AL/QL)	0.141
Avg NBT % (Dbl AL/Mth)	0.120
Avg NBT % (Dbl QL/Mth)	0.111
Avg Gr12 & NBT	0.016
Avg Gr12 & NBT (Dbl Gr12 Mth)	-0.012
Avg Gr12 & NBT (Dbl Gr12 Mth & Sci)	-0.048

NBT scores have a higher correlation with CSC1015F grades compared to Grade 12 results in general, with the highest correlation between admissions data and CSC1015F results found when an average of all grade 12 results (that are incorporated into this study) and NBT scores is used as a means of benchmarking students. However this means of benchmarking has the smallest variance as a feature.

Two benchmarking methods stand out: using straight *NBT QL* a *NBT Mth* scores as benchmark datasets. These datasets display a relatively high standard deviation

in terms of percentages ($> 13\%$) and correlate better with CSC1015F grades than many other benchmarks (with $r = 0.53$ and $r = 0.51$ respectively)

Three scatter plots of grade/benchmark scores are included as a visual representation of the relationship between benchmark variance and correlation: Figure 6.1 shows a plot of *GR12 ENG* benchmark vs CSC1015F scores, Figure 6.2 shows a plot of *Avg Gr12 & NBT* benchmark vs CSC1015F scores, and Figure 6.3 shows a plot of *NBT QL* vs CSC1015F scores.

No correlation is found between student performance and Sakai usage. This is perhaps because the event data doesn't contain course codes, and so it is only possible assess the count of presence events on all sites in relation to the CSC1015F result and not usage of the CSC1015F Sakai site specifically. An example of a scatter plot of rank change vs Sakai events count is shown in Figure 6.4.

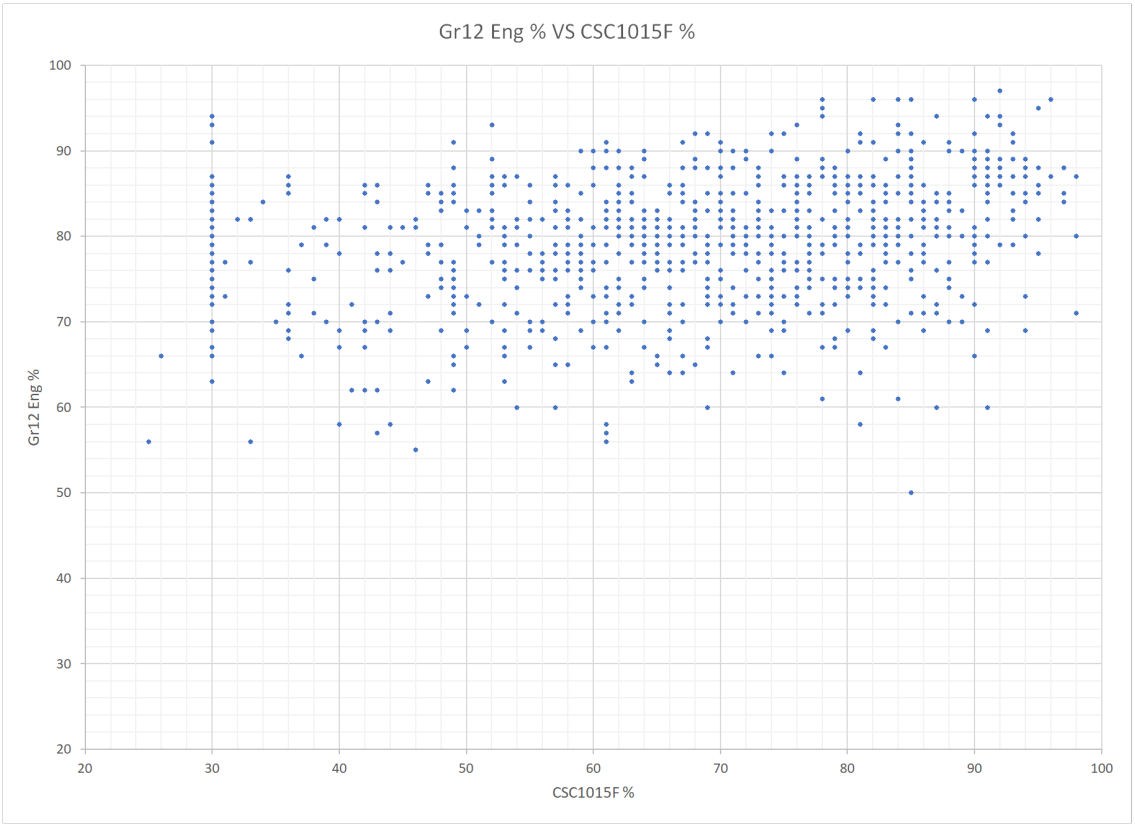


Figure 6.1: CSC1015F % vs Gr12 Eng %

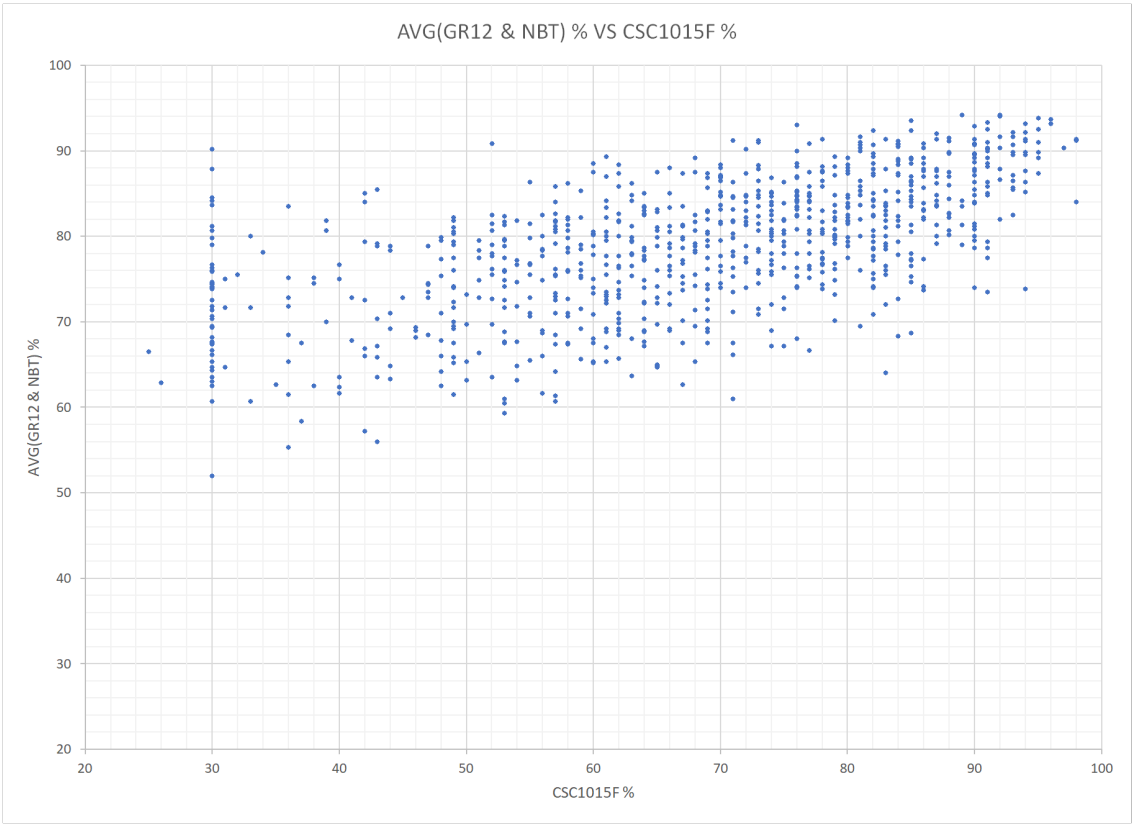


Figure 6.2: CSC1015F % vs AVG(GR12 & NBT) %

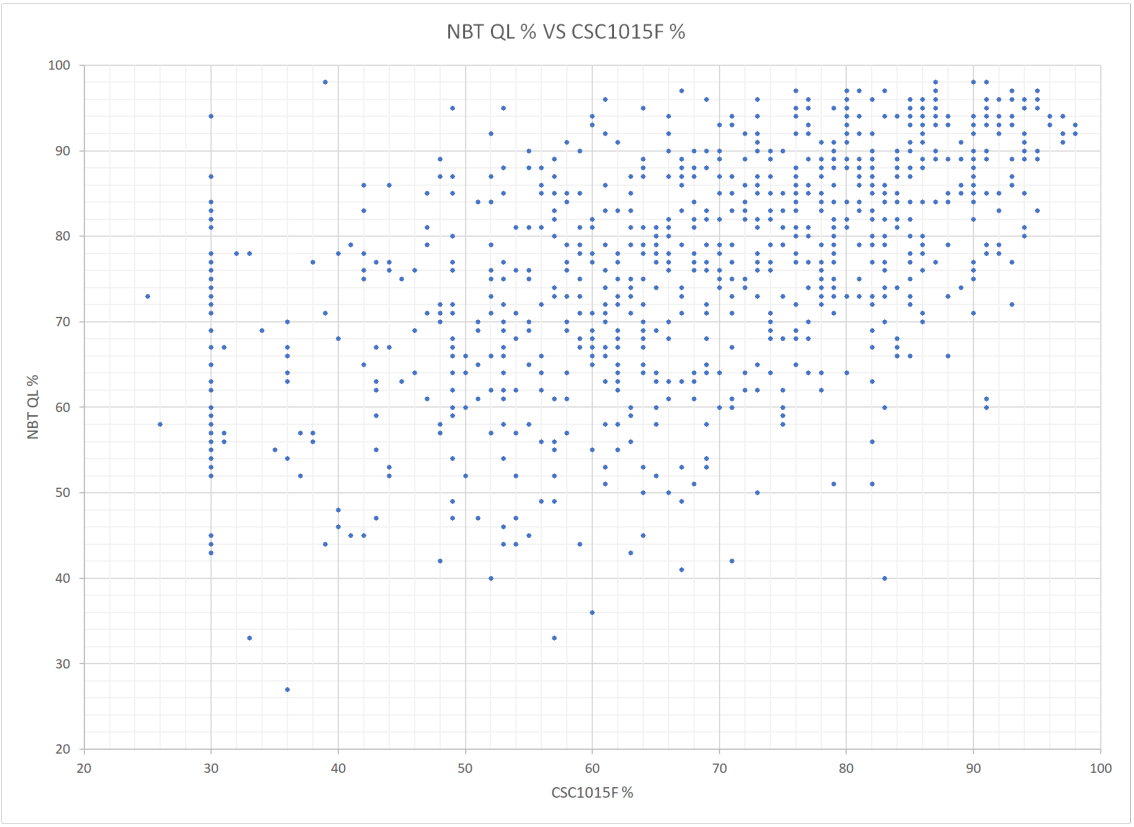


Figure 6.3: CSC1015F % vs NBT QL %

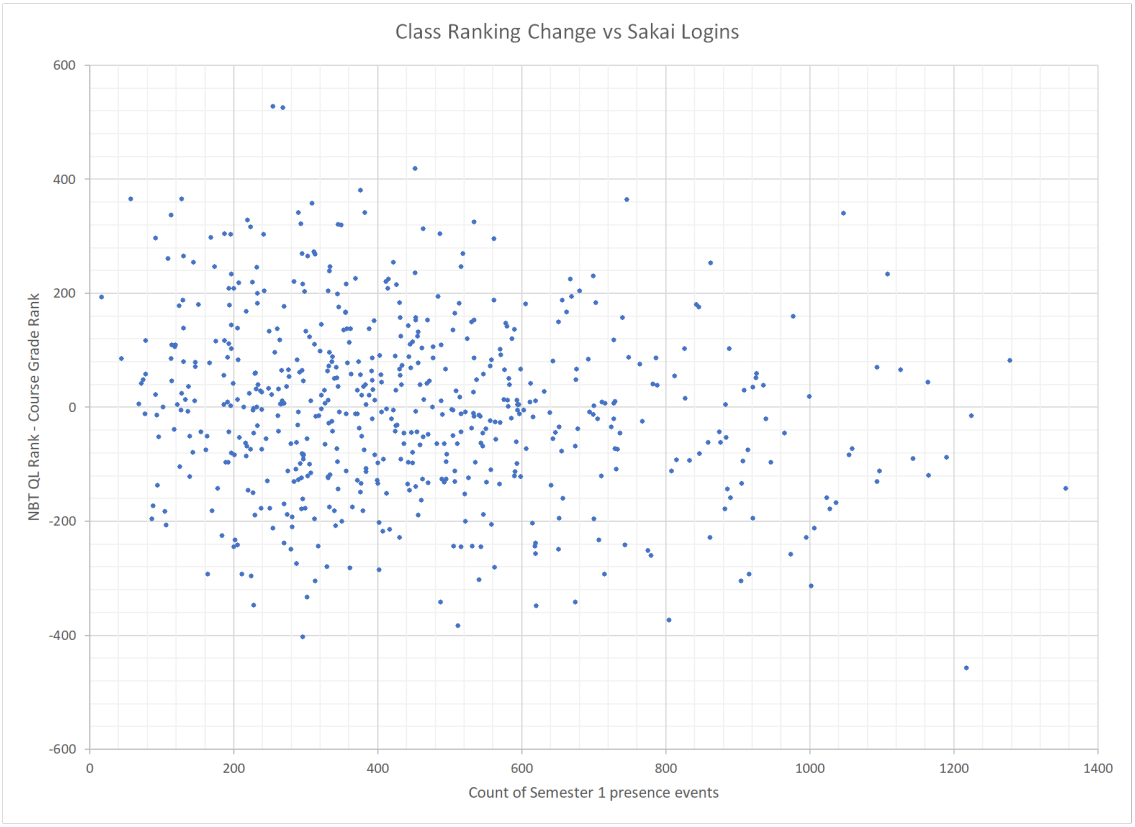


Figure 6.4: Δ Grade/NBT QL class rank vs Sakai events count

Chapter 7

Conclusion

This chapter summarises the work presented and suggests some avenues for future research.

7.1 Summary

CouchDB uses MapReduce as a means of producing indices via parallel processing, and thus requires map functions that operate on a single document at a time. As such, performing cross-document joins and aggregations on datasets using CouchDB's implementation of MapReduce requires architecture-wide considerations. This is in contrast to relational database systems, where operation implementation is transparent to users and developers.

Document-oriented stores such as CouchDB use nesting to ensure that a single document represents a single real-world entity wherever possible. However, particularly in a distributed system, there are frequently situations where, rather than embedding related entities, separate documents are used instead. One advantage of separation over nesting is that it prevents documents from becoming too large, which would make document transfer too costly. Another is that data received from e.g. a data lake is typically in separate, flattened structures rather than hierarchical/embedded format, and much more easily loaded into CouchDB as is. It is when separate documents are used that the need for inter-document joins and aggregations arises.

This thesis has shown that join and aggregation can be performed on such data in the context of CouchDB using the following approach:

1. A configurable framework is provided for ETL (extract - transform - load)

which not only simplifies ETL processes but also runs such tasks concurrently and asynchronously in order to improve performance

2. MapReduce is used as a means of normalizing entity representation, performing aggregations and indesing entity representations based on join-keys
3. Joins are then performed using these indices
4. Where joins are required on projection- and selection-based subsets of data, the ETL framework is used to create such subsets beforehand, so as to improve join performance

Although indices may be time-consuming to calculate initially, scanning B+trees is very efficient, and CouchDB indices are updated incrementally rather than re-created, so both data retrieval and joins performed via index scans are very efficient.

The above approach was applied to 3 data sets provided by the University of Cape Town, which varied in size from 12 219 records to over 44 million records. The ETL, join and aggregation operations were correctly computed and executions ran in reasonable time using only 8 shards and a single computer as the server. While this case study was primarily used as a proof of concept, some of the results obtained are of interest to those focussing on educational data mining research. In particular, the emergence of National Benchmark Test scores as more indicative of CSC1015F performance than Grade 12 scores is a result worth investigating further.

7.2 Future Work

CouchDB views are optimized when using built-in reduce functions, with custom reduce functions performing most poorly on Windows machines. As this project was completed on the Windows OS, the analysis on how best to aggregate the different entities was confined to using just the built-in reduce functions.

It may be worth investigating using custom reduce functions for MapReduce jobs. This would allow for much more varied map function output structure and provide a means of performing joins during index reduction instead of on index scanning. Re-

duce function calculation represents a small percentage of computer resource usage overall [32], and different environment configurations provide different performance contexts in terms of what overheads are/are not acceptable. And in any case, a system that utilizes CouchDB is likely to be based on a cluster of Linux machines rather than a single Windows machine.

Feasibility rather than performance was the focus of this work; if implemented on a network cluster of computers, performance and scalability could be investigated. It would be worth investigating the benefit of clustering CouchDB across many separate nodes with varying configurations for replication and data redundancy (node copies). CouchDB is configured to use 8 shards by default (even on a single server) and processes data in parallel (across 2 physical cores for this project), it is likely that deploying shards to separate servers would greatly increase performance and decrease indexing time. CouchDB disperses documents evenly across shards in a random fashion, suggesting that the workload of indexing the documents would be distributed evenly across all the shards of the database [42]. It is likely sharding would benefit large datasets, but not smaller datasets since the cost of network interactions would also increase if shards were distributed across separate nodes.

There is further scope to test implementing relational operations in a similar software stack that were not implemented in this project. These are: *division*, *set union*, and *set difference*.

There is also scope to develop a GUI for the *nETL* application.

References

[1] Apache.org. *6.1.1. View Functions*. URL: http://docs.couchdb.org/en/stable/ddocs/ddocs.html#view-functions (visited on 04/09/2018).

[2] Apache.org. *CouchDB Blog (news): 2.0*. Sept. 20, 2016. URL: https://blog.couchdb.org/2016/09/20/2-0/ (visited on 02/14/2018).

[3] Apache.org. *CouchDB Blog (news): How CouchDB enabled eHealth Africa to collect and sync data in the field*. July 5, 2017. URL: https://blog.couchdb.org/2017/07/05/how-couchdb-enabled-ehealth-africa-to-collect-and-sync-data-in-the-field/ (visited on 02/14/2018).

[4] Apache.org. *CouchDB Docs: API*. URL: http://docs.couchdb.org/en/2.1.1/api/index.html (visited on 02/14/2018).

[5] Apache.org. *http://couchdb.apache.org/*. URL: http://couchdb.apache.org/ (visited on 04/09/2018).

[6] Apache.org. *Introduction to CouchDB Views*. Wiki: last edited 05/06/2013. URL: https://wiki.apache.org/couchdb/Introduction_to_CouchDB_views (visited on 02/14/2018).

[7] Apache.org. *Powered by Apache Hadoop*. Wiki: last edited 12/07/2017. URL: https://wiki.apache.org/hadoop/PoweredBy (visited on 02/14/2018).

[8] Apache.org. *Using Fauxton*. URL: https://couchdb.apache.org/fauxton-visual-guide/index.html#_config (visited on 02/15/2018).

[9] Atzeni, P. et al. "Data modeling in the NoSQL world". In: *Computer Standards And Interfaces* (2016). ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2016.10.003. URL: http://www.sciencedirect.com/science/article/pii/S0920548916301180.

[10] Baker, R. and Yacef, K. "The State of Educational Data Mining in 2009: A Review and Future Visions". In: *Journal of Educational Data Mining*. Vol. 1. Jan. 2009, pp. 3–17.

[11] Balestra, M. *Predicting Killer Course Student Performance Using Decision Trees*. Honours Project: University of Cape Town. 2017.

[12] Berman, S. pers. comm. (Associate Professor UCT). 2018.

[13] Bower, E. *Node.js generator-based line reader*. Source code (MIT License). URL: https://github.com/neurosnap/gen-readlines (visited on 02/14/2018).

[14] Bray T., E. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. RFC Editor, Mar. 2014. URL: https://www.rfc-editor.org/info/rfc7159.

[15] Casper, D. *Predicting whether students will finish their degree within minimum time through the use of Decision Trees*. Honours Project: University of Cape Town. 2017.

[16] Chandar, J. *Join Algorithms using Map/Reduce*. MA thesis: University of Edinburgh. 2010.

[17] Chandra, D. G. "BASE analysis of NoSQL database". In: *Future Generation Computer Systems* 52.Supplement C (2015). Special Section: Cloud Computing: Security, Privacy and Practice, pp. 13–21. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2015.05.003. URL: http://www.sciencedirect.com/science/article/pii/S0167739X15001788.

[18] Couchapp Organisation. *Utilities to make standalone CouchDB application development simple*. Source code (Apache 2.0 License). URL: https://github.com/couchapp/couchapp (visited on 02/14/2018).

[19] Couchbase. *Understanding custom reduces and rereduce*. URL: https://developer.couchbase.com/documentation/server/3.x/developer/dev-guide-3.0/reduce-rereduce.html (visited on 04/09/2018).

[20] Couchbase. *Why NoSQL Database?* URL: https://www.couchbase.com/resources/why-nosql (visited on 02/14/2018).

[21] Dean, J. and Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[22] Dimitris, K. and Christos, P. "Analyzing Student Performance in Distance learning with Generic Algorithms and Decision Trees". In: *Applied Arti-*

ficial Intelligence 20.8 (2006), pp. 55–674. DOI: 10.1080/08839510600844946. eprint: http://dx.doi.org/10.1080/08839510600844946. URL: http://dx.doi.org/10.1080/08839510600844946.

[23] ECMA INTERNATIONAL. *ECMAScript® 2017 Language Specification (ECMA-262, 8th edition, June 2017)*. URL: https://www.ecma-international.org/ecma-262/8.0/index.html (visited on 03/15/2018).

[24] ECMA INTERNATIONAL. *ECMAScript® Language Specification (ECMA-262, 5.1 edition, June 2011): Executable Code and Execution Contexts*. URL: https://www.ecma-international.org/ecma-262/5.1/#sec-10 (visited on 04/09/2018).

[25] FOWLER, M. *MartinFowler.com: AggregateOrientedDatabase*. Jan. 19, 2012. URL: https://martinfowler.com/bliki/AggregateOrientedDatabase.html (visited on 02/14/2018).

[26] FOX, A. and BREWER, E. A. "Harvest, Yield, and Scalable Tolerant Systems". In: *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*. HOTOS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 174–. ISBN: 0-7695-0237-7. URL: http://dl.acm.org/citation.cfm?id=822076.822436.

[27] GOLDBERG, D. "What Every Computer Scientist Should Know About Floating-point Arithmetic". In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. URL: http://doi.acm.org/10.1145/103162.103163.

[28] IEEE. *IEEE Standard for Floating-Point Arithmetic*. Tech. rep. Aug. 29, 2008, pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[29] KHAN, Z. N. "Scholastic Achievement of Higher Secondary Students in Science Stream". In: *Journale of Social Sciences* 1.2 (2005), pp. 84–87. ISSN: 1549-3652.

[30] LEHNARDT, J. pers. comm. (Apache CouchDB Vice President). Nov. 2, 2017.

[31] LEHNARDT, J. pers. comm. (Apache CouchDB Vice President). Oct. 25, 2017.

[32] LEHNARDT, J. pers. comm. (Apache CouchDB Vice President). Nov. 1, 2017.

[33] LEHNARDT, J. pers. comm. (Apache CouchDB Vice President). Feb. 28, 2018.

[34] Lotfy, A. E. et al. “A middle layer solution to support ACID properties for NoSQL databases”. In: *Journal of King Saud University - Computer and Information Sciences* 28.1 (2016), pp. 133–145. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2015.05.003. URL: http://www.sciencedirect.com/science/article/pii/S1319157815001044.

[35] Media, O. *CouchDB The Definitive Guide (draft)*. URL: http://guide.couchdb.org/draft/btree.html (visited on 02/14/2018).

[36] Microsoft. *SQL Server Integration Services*. URL: https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services (visited on 03/15/2018).

[37] Mierle, K. et al. “Mining Student CVS Repositories for Performance Indicators”. In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1082983.1083150. URL: http://doi.acm.org/10.1145/1082983.1083150.

[38] MikeMcl. *An arbitrary-precision Decimal type for JavaScript*. Source code (MIT License). URL: https://github.com/MikeMcl/decimal.js/ (visited on 02/14/2018).

[39] Mocha Organisation. *simple, flexible, fun javascript test framework for node.js & the browser*. Source code (MIT License). URL: https://github.com/mochajs/mocha (visited on 02/14/2018).

[40] Mozilla Foundation. *Iterators and generators*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators (visited on 03/10/2018).

[41] Mozilla Foundation. *Promise*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (visited on 03/21/2018).

[42] Newson, R. and Lehnardt, J. pers. comm. (Apache CouchDB). Nov. 7, 2017.

[43] Node.js. *About Node.js®*. URL: https://nodejs.org/en/about/ (visited on 03/15/2018).

[44] npm, Inc. *npm*. URL: https://www.npmjs.com/ (visited on 04/05/2018).

[45] Al-Radaideh, Q. A., Al-Shawakfa, E. M., and Al-Najjar, M. I. "Mining Student Data Using Decision Trees". In: *The 2006 International Arab Conference on Information Technology* (Nov. 1, 2017).

[46] Rajaraman, A. and Ullman, J. D. *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011. Chap. 2. isbn: 1107015359, 9781107015357.

[47] Rashmi, K. et al. "Development of a Middleware Layer for CouchDB NoSQL System for Providing Transactional Properties". In: *International Research Journal of Engineering and Technology (IRJET)* 4.4 (Apr. 4, 2017), pp. 3220–3223. issn: 2395-0056.

[48] Request Organisation. *request*. Source code (Apache 2.0 License). url: `https://github.com/request/request` (visited on 02/14/2018).

[49] Sadalage, P. J. and Fowler, M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 1st. Addison-Wesley Professional, 2012. isbn: 0321826620, 9780321826626.

[50] sakaiproject.org. *Sakai*. url: `https://www.sakaiproject.org/` (visited on 02/15/2018).

[51] Simpson, K. *You-Dont-Know-JS:Scope & Closures*. Sebastopol, CA USA: O'Reilly Media, 2014-03. Chap. 2.

[52] Simpson, K. *You-Dont-Know-JS:this & object prototypes*. Sebastopol, CA USA: O'Reilly Media, 2014-07. Chap. 6.

[53] talend. *Data Integration Products*. url: `https://www.talend.com/products/data-integration/` (visited on 03/15/2018).

[54] zapier. *Connect Your Apps and Automate Workflows*. url: `https://zapier.com/` (visited on 03/15/2018).